MINICOMPUTERS
theory & operation

Donald Eadie

Minicomputers: theory and operation

Donald Eadie

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

To my grandchildren:
Cara, Christopher, Diana, and Patrick Willis

Library of Congress Cataloging in Publication Data

Eadie, Donald.
Minicomputers, theory and operation.

Includes bibliographical references and index.
1. Minicomputers. I. Title.
QA76.5.E16 001.6'4'04 79-10973
ISBN 0-8359-4387-9

© by
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

All rights reserved. No part of this book may be reproduced in any way, or by any means, without permission in writing from the publisher.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Contents

Preface, ix

1 History and Processors in Society, 1

1-1 Introduction, 1
1-2 The Beginnings, 1
1-2.1 Abacus, 1; 1-2.2 Napier's Bones, 1;
1-2.3 Early Calculating Machines, 2; 1-2.4 Babbage's Work, 2; 1-2.5 Continued Development of Calculating Machines, 2
1-3 The Punched-Card Era, 2
1-4 Computer Development Before and During World War II, 5
1-4.1 Analog Scientific Calculations, 5; 1-4.2 Digital Scientific Calculations, 6
1-5 Data Processors and Society, 7
1-5.1 General, 7; 1-5.2 Control Processing, 8;
1-5.3 Time Share Processing, 9; 1-5.4 Communications Processing, 10; 1-5.5 Specialized Processing, 11;
1-5.6 Minicomputers, 11
1-6 Summary, 13
1-7 Problems, 13

2 Fundamentals, 15

2-1 Introduction, 15
2-2 Number Systems, 15
2-2.1 Representation, 15; 2-2.2 Binary, 17
2-3 Conversion—Base to Base, 18
2-4 Coded Numbers (BCD), 21
2-5 Coded Numbers and Letters, 21
2-6 Binary Arithmetic, 25
2-7 Complements, 28
2-8 Signed Binary Numbers, 31
2-9 Octal Number Systems, 35
2-10 Hexadecimal (HD) Number Systems, 36
2-11 Conversions, 38
2-12 Summary, 38
2-13 Problems, 38

3 Binary Control and Data, 40

3-1 Introduction, 40
3-2 Data in Binary Systems, 40
3-3 Program Components, 42
3-3.1 Instructions, 42; 3-3.2 Constant and Variable Data, 43
3-4 Computer Components and Systems, 43
3-5 Summary, 43
3-6 Problems, 44

4 Hardware Logic Block Elements, 45

4-1 Introduction, 45
4-2 Logic Hardware Forms, 45
4-3 Logic Gates, 46
4-4 Flip-Flop, 48
4-4.1 Flip-flop Controlled AND Function Gates, 49
4-5 Counters, 50
4-6 Registers, 50
4-7 Combinational Circuits Composed Primarily of Gates, 51
4-8 Interfacing, 54
4-9 Conversion of Logic Types, 55
4-10 ROMS and p/ROMS as Logic Elements, 57
4-11 Bidirectional Bus Drivers/Receivers, 58
4-12 Summary, 58
4-13 Problems, 60

5 Binary Processor Organization, 61

5-1 Introduction to Binary Processor Organization (First Generation Minicomputer), 61
5-2 Memory, 62
5-2.1 Core, 62; 5-2.2 Semiconductor Memories, 64
5-3 Central Processing Units (CPU), 64
5-4 Arithmetic Unit Operation, 67
5-4.1 Addition/Subtraction in Binary Machines, 67;
5-4.2 General Multiplication/Division, 67;
5-4.3 Double-Precision Arithmetic, 68; 5-4.4 Floating-point Operations, 69
5-5 Control Operations, 72
5-5.1 Instruction Decoding, 72; 5-5.2 Instruction Counter, 75; 5-5.3 Usual Control vs. Microprogramed Control, 76
5-6 Extension of Address Capability, 76
5-7 The Complete Processor, 77
5-8 Input/Output 78
5-9 Methods of Interfacing Computers and Peripherals (H316), 80
5-9.1 DIO Approach, 80
5-10 DMA/DIO Control, 83
5-11 Summary, 83
5-12 Problems, 84

6 Peripherals, 86

6-1 General, 86
6-2 Loading and Unloading Devices, 86
6-2.1 Paper Tape Devices, 86; 6-2.2 Card Devices, 87; 6-2.3 Incremental Tapes, 88
6-3 Display Devices, 88
6-3.1 Printers, 88; 6-3.2 CRT Displays, 90; 6-3.3 Plotters, 90
6-4 Bulk Store Memory Peripherals, 91
6-4.1 General, 91; 6-4.2 Continuous-run Magnetic Tapes (CRMT), 92; 6-4.3 Magnetic Drums, 94; 6-4.4 Disk Files, 94; 6-4.5 Comparison of the Magnetic Bulk Stores, 95; 6-4.6 Cassettes, 97; 6-4.7 Charge Coupled Devices (CCD) Bulk Store, 98; 6-4.8 Magnetic Domain Bulk Store, 99
6-5 Control System Interfacing, 100
6-5.1 A/D and D/A Converters, 100; 6-5.2 Analog Interfacing, 102; 6-5.3 Digital Interfacing, 103
6-6 Summary, 104
6-7 Problems, 104

7 Instructions and Programming, 105

7-1 Introduction, 105
7-2 Fields of Instruction, 106
7-2.1 Memory Reference Instructions, 107; 7-2.2 Shift Instructions, 107; 7-2.3 Generic Instructions, 108; 7-2.4 Input/output (I/O) Instructions, 109; 7-2.5 Optional Equipment Instructions, 109; 7-2.6 Data Representation, 110
7-3 The Instruction Repertoire, 110
7-4 Instruction Organization, 111
7-4.1 Single Address Instructions, 111; 7-4.2 Multiple Address Instructions, 112
7-5 Problem Analysis in Programming, 113
7-6 Programming Steps, 115
7-6.1 Sample Instructions, 115; 7-6.2 Problem Solving and Programming, 115; 7-6.3 Symbolic Notation and Flow-Charting, 116
7-7 Decision Blocks and Their Use to Control Programs, 119
7-8 Subroutines, 122
7-9 Summary, 122
7-10 Problems, 123

8 Programming Aids and Software, 125

8-1 Introduction, 125
8-2 Types of Programs, 127
8-2.1 Operational Programs, 127; 8-2.2 Executive Programs,

127; 8-2.3 Utility Programs, 128;
8-2.4 Simulators, 129; 8-2.5 Assemblers and Compilers, 129; 8-2.6 Generator Programs, 133
8-3 Debugging Programs, 133
8-4 Compilers and Assemblers, 134
8-4.1 General, 134; 8-4.2 Assembler, 135;
8-4.3 Macroinstructions, 139; 8-4.4 Assemblers and Macroinstructions, 139
8-5 Compilers, 140
8-6 Subprograms, 142
8-7 Software, 143
8-7.1 Manufacturer Supplied Software, 145
8-8 Allocation of Memory During Operation, 147
8-9 A Language for Time-Sharing—Basic, 148
8-9.1 General, 148; 8-9.2 Sample Program, 150
8-10 Summary, 151
8-11 Problems, 152

9 FORTRAN and Other Compilers, 154

9-1 Introduction, 154
9-2 Number and Constant Representation, 156
9-3 Input/Output Statements, 156
9-3.1 Read and Write Statements, 158; 9-3.2 End, Stop, and Pause Statements, 159
9-4 Arithmetic (Or Assigned) Statements, 160
9-5 Control Statements, 161
9-5.1 GO TO Statement, 161; 9-5.2 IF Statement, 162
9-6 Specification Statements, 165
9-7 Subprogram Statements, 167
9-8 Conversational Fortran, 169
9-9 Limitations to Fortran, 170
9-10 COBOL, 171
9-11 Summary, 174
9-12 Problems, 175

10 Minicomputers (First Generation), 176

10-1 Introduction, 176
10-2 Characteristics of Minicomputers, 176
10-2.1 Price Range, 176; 10-2.2 Organization, 176; 10-2.3 Instruction words, 177; 10-2.4 Software, 177; 10-2.5 Fabrication, 179; 10-2.6 Limited Service Provided by the Manufacturer, 180
10-3 Organization Example, 180
10-3.1 H316 Computer, 180; 10-3.2 Internal Organization, 181
10-4 Options Available, 182
10-4.1 High-Speed Arithmetic (HSA), 182;
10-4.2 Real-Time Clock, 183; 10-4.3 Additional Memory, 183; 10-4.4 Direct Multiplex Control (DMC), 183; 10-4.5 Peripheral Devices, 183; 10-4.6 Priority Interrupts, 183

10-5 Using the Assembler, 184
10-6 Control Panel Switches, 185
10-6.1 Control Panel Switches Listing, 185;
10-6.2 Entering Code in Memory and Other Registers, 186
10-7 Use in Control Applications, 188
10-8 Interfacing Problems and How to Avoid Them, 188
10-9 Minis Compared, 188
10-10 Summary, 190
10-11 Problems, 191

11 The Interdata 7/16 Computer, 192

11-1 System Architecture (CPU), 192
11-1.1 General, 192; 11-1.2 Memory, 192; 11-1.3 Standard Software, 192; 11-1.4 Instruction Set, 193; 11-1.5 Input/Output, 194; 11-1.6 Peripheral Products Available, 194; 11-1.7 Microprogram Control, 194; 11-1.8 The Program Status Word (PSW), 194; 11-1.9 Other Compatible Processors of the INTERDATA Family, 195
11-2 The Hexadecimal (HD) Display Panel, 196
11-3 Instruction Formats, 197
11-3.1 General, 197; 11-3.2 Format Details, 197;
11-3.3 Instructions of the Data Manipulating Class, 199
11-4 Branching, 199
11-4.1 General, 199; 11-4.2 Branching Instructions, 199
11-5 Fixed-POINT Arithmetic, 210
11-5.1 General, 210; 11-5.2 Arithmetic Instructions, 210
11-6 Floating-Point Arithmetic, 210
11-6.1 General, 210; 11-6.2 Exponent Overflow and Underflow, 214; 11-6.3 Guard Digit and Rounding, 214; 11-6.4 Floating-Point Instructions, 214
11-7 Status Switching and Interrupts, 214
11-7.1 General, 214; 11-7.2 PSW Bit Functions, 215; 11-7.3 Processor Interrupts, 215;
11-7.4 Reserved Memory Locations, 215; 11-7.5 Status Switching and Interrupt Instructions, 215
11-8 Sample Programs, 221
11-8.1 General, 221; 11-8.2 Program Examples, 221; 11-8.3 Simple Add-Subtract, Multiply Problem, 221; 11-8.4 Fixed-Point Add Test Program, 222;
11-8.5 Divide (DH) Test Program, 222; 11-8.6 Branch on Index High Test Program, 222; 11-8.7 Branch on Positive (BP) Test Program, 223; 11-8.8 Compare Logical Halfword Test Program, 224; 11-8.9 Branch-on-Not-Equal Register (BNER) Test Program, 224;
11-8.10 Subtraction Halfword Reg (SHR) Test Program, 224; 11-8.11 Branch and Return from Subroutine Program, 225; 11-8.12 Program to Load Display with Bytes Test Program, 225; 11-8.13 Load and Add Floating-Point Numbers, 226
11-9 Summary, 226
11-10 Problems, 227

12 Microprocessors and Microcomputers, 229

12-1 Introduction, 229
12-2 Microprocessor Stacks, 230
12-3 A Typical Microcomputer, 231
12-4 Microprocessor Features, 232
12-5 INTEL 8080A, 233
12-6 Motorola M6800, 235
12-7 Sixteen-Bit Microprocessors, 237
12-7.1 General, 237; 12-7.2 A Sixteen-Bit Minicomputer of Microprocessor Components, 238
12-8 Examples of Microcomputer Applications, 241
12-9 Summary, 242
12-10 Problems, 242

13 Typical Applications of Minicomputers, 243

13-1 Introduction, 243
13-2 Automated Testing, 243
13-3 The Disney World Monitor, 244
13-4 Power Systems, 245
13-4.1 General, 245; 13-4.2 Minicomputer Control of a Combined-Cycle Power Plant, 245
13-5 Wall Street Automation, 249
13-6 Microcomputer/Minicomputer Applications, 249
13-7 Summary, 249
13-8 Problems, 250

Appendix A, Answers to Even-Numbered Problems, 251
Appendix B, Characteristics of 8080A System, 260
Appendix C, Bibliography, 262
Appendix D, 8080A Instruction List, 265
Appendix E, M6800 Instruction List, 269

Index, 271

Preface

This book is intended to introduce all fundamentals of data processing. The first three chapters lay the foundation. The remainder of the book burrows into the more specialized areas unique to the field. In the past it has been customary for practitioners to specialize primarily in one area—hardware design, analysis, system design, programming, or data processing. This approach has led to isolation of people in different specialties. Even today communications can be strained. There is much effort with learning a specialty; thus, it is understandable that a person thinks in terms of the area he understands best, and feels strange outside that area. No one person knows all there is to know about data processing and the associated specialties—especially in light of the explosion of knowledge that has occurred over the past twenty years.

The object of this writing is to unite the specialties—at least on an introductory level—in the hope that the specialties will blend more into a total picture. In this way it is hoped the reader can decide which specialty, if any, he might feel inclined to study further. Today it is most important that a user be able to translate his particular application into terms to which those who are responsible for system design (hardware and software) can relate. When the user defines what he wants, he becomes a vital participant in the system design process. He will be able to see if his desires can be built into a successful finality, understanding which of the specified requirements are achievable in a realistic manner. The development of an entirely new system is not without technical and financial risk, but these are controllable to a large degree.

This book places considerable emphasis on the minicomputer because of its wide application since 1968. This class of machine is widely versatile, relatively easy to apply, and replaces many of the more costly machines of earlier vintage. However, technology advances in an unabated sequence. In 1972 the microcomputer was announced. This is a small computer consisting generally of a few LSI packages usually mounted on a single small plug-in board. This approach dramatically lowers the eventual cost of minimum computers and extends their

operating range downward into almost every nook and cranny of control applications.

The book starts out by emphasizing the elements of data processing. After introductory Chapter 1, Chapter 2 continues with codes employed in data processing, number systems generally, with further emphasis on binary, octal, and hexidecimal systems, which are used universally in computer systems. Chapter 3 outlines forms of binary data and commands, and introduces the general computer block diagrams. Chapter 4 explains digital logic, combinational digital elements, and some of the circuits used for system interfacing. Chapter 5 continues with an introduction how a simple minicomputer functions, and closes with a discussion of various interfacing methods employed today in minicomputers and microcomputers. Chapter 6 introduces peripheral devices of various types and concludes with examples of control system interfacing methods (multiplexors, converters, and direct digital interfacing). Chapters 7 and 8 discuss typical instruction formats, programming, and software with an introduction to the interpreter BASIC, which was used originally for time-share systems, but now has effectively been adopted as a compiler like FORTRAN. Chapter 9 briefly outlines FORTRAN and COBOL as examples. Chapter 10 describes the Honeywell H316 minicomputer—an example of an early version, while Chapter 11 explores the Interdata 7/16, a fourth-generation minicomputer with the capability of the large-scale processors of a decade earlier. Chapter 12 introduces a few of the microcomputers at the moment among the most used. This chapter was included because the two fields—microprocessors and minicomputers—are rapidly merging. Chapter 13 lists a few sample applications of minicomputers and microcomputers. Microcomputers are generally used where 16-bit minicomputers are not needed, and too costly to apply in an underused application.

Thus, Chapters 1 through 9 treat the fundamentals of data processing. Chapters 10 through 13, on the other hand, discuss typical minicomputer and microcomputer hardware and their application. In reality there are two sections to the book which together are probably too extensive to complete in a one-semester course. The teacher will have to decide which topics to treat and which to ignore if the course is to be of standard length. There is a lot of detail—enough to spend two semesters of work.

Practice problems are included at the end of each chapter. Answers for the even-numbered problems are included in Appendix A at the back of the book. These answers should allow the student to check his own work.

My grateful thanks to those people who encouraged me to complete this work, including my wife Ruth, my brother Stuart, and my

many associates at Aero-Florida Division of Honeywell Inc. Special thanks are due Mrs. Lori McMahon, who typed the majority of the manuscript, and to Mr. Matthew I. Fox, president of Reston Inc., who allowed me to complete a work he expected from me much in advance of this date. Thanks are extended to the staff at Hillsborough Community College, especially Professor Arnold Kroeger. Also I thank Mr. Barton Hoeg, Manager of Editorial Services of Interdata Inc. for supplying photos for Chapter 11.

Donald Eadie
Belleair Bluffs, Florida

1

History and Processors in Society

1-1 INTRODUCTION

This chapter introduces the reader to the subject of data processing. It is rather general in its discussion, and for specific details on the persons and devices mentioned, the reader is referred to more detailed treatments appearing in other publications. A few of the events of historical significance are mentioned to illustrate the growth in ideas leading to the modern computer.

The computer is really an evolution rather than an invention of any one person. Not only have the mechanisms themselves evolved, but so have the applications of how a processor fits into a system. Although *batching* and *control* applications were conceived rather early in the computer explosion following World War II, it was a few years before experiments in *time sharing* began in earnest. This chapter discusses in an introductory way many of these topics.

1-2 THE BEGINNINGS

1-2.1 Abacus

The advantages of instruments to assist with computations were recognized in antiquity. The Abacus, consisting of several rows of beads on a frame, permitted merchants in early Egypt and Babylon to compute transactions beyond the range of their individual fingers and toes. The Abacus is still employed today in many parts of the world.

1-2.2 Napier's Bones

In the early 1600s, John Napier of Scotland invented a device that assisted greatly in the use of logarithms to perform arithmetic operations such as multiplication and long division. This device became known as

"Napier's bones." Today, the descendant device, employed so often by engineers and others, is the *slide rule*.

1-2.3 Early Calculating Machines

Calculating machines, first employed historically by Blaise Pascal in 1642 and by Von Leibniz a few years later, defined techniques that have been improved through the centuries and led directly to calculation approaches employed by processors designed in the past 20 years.

1-2.4 Babbage's Work

It is to Charles Babbage, a nineteenth-century English mathematician, that the title *"Father of the Modern Computer"* generally applies. His earnest, but largely unsuccessful, effort to develop two major computing machines for British government departments in the early to mid-1800s outlined generally the basic organization of a modern processor. The *memory*, the *arithmetic processor*, and the *input/output* are identifiable in his descriptive writings. Also, he proposed the idea of storing in the form of commands the operation sequence to be performed on the data. However, the technology of his era was not up to the task of mechanizing successful hardware. Nevertheless, a hundred or so years later, his work was studied before the design of the Harvard Machines (MARK I, etc.) was embarked upon.

1-2.5 Continued Development of Calculating Machines

Such improvements as *keyboards*, *punch paper tapes*, and *punched cards* continued to be added from time to time to the early designs. In fact, the machines that Babbage planned were eventually built in Sweden and the United States. In the late 1800s, Hollerith invented the *punched card*, and this invention was the initial impetus behind the establishment of the office machine industry that mushroomed so rapidly in the first half of the twentieth century.

1-3 THE PUNCHED-CARD ERA

The punched card invented by Herman Hollerith about 1890, inaugurated the era of machine data processing. Machines soon were designed to punch, read, verify, and sort punched cards. Thus, it was possible to record data on a standard form that could be manipulated by machines, and such routine processes as "searching" for an item among

stacks of items, sorting the items in a preordered sequence, and obtaining tabulations could now be performed by electromechanical machinery rather than by the manual process of eyeballing records, i.e., studying columns of figures. This era extended uninterrupted into the early 1940s. In the later years, even more talented card-handling machines manipulated the data, for example, permitting arithmetic calculations (addition, multiplication, division, etc., on the data) and providing the capability to print out business reports. It was the existence throughout the nation and the world of vast quantities of card-handling equipment and the decision to use it for inputting data to a computer or reading out data from a computer that gave IBM Corporation such a natural jump on its competitors in the 1950s. Many potential processor customers were already familiar with the IBM equipment, had been serviced well by the vast force of IBM field servicemen, and naturally, because of past experience, favored systems supplied by IBM. The card readers and card punches have survived into the electronic data processing (EDP) era of the 1970s in substantially their original role. Such familiar operations as sorting, tallying, and calculating are less likely to be done by the mechanical equipment nowadays on the card decks themselves, but the processor performs the data manipulation on the data once they have been committed to the processor's memory. Then, if the final results are required to be stored on cards, these can be provided as a readout form by a card punch peripheral. One might alternately consider readout in the form of a magnetic tape record, or perhaps a printed page, for example. Most modern computer installations provide a variety of options on how to represent the final output results.

Figure 1-1 illustrates the Hollerith card keypunch machine. Blank

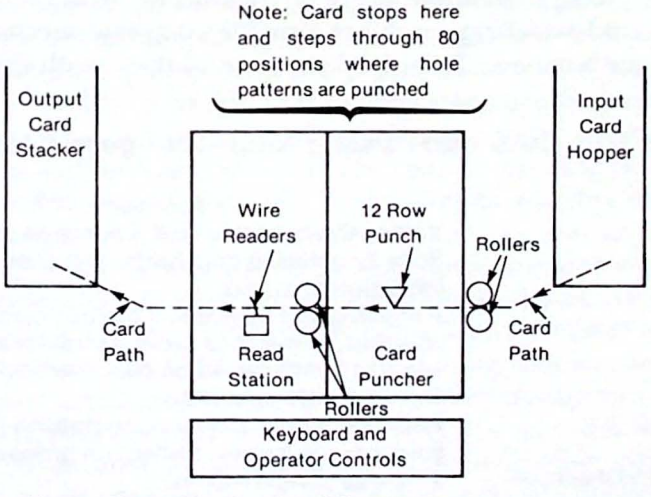


FIGURE 1-1. Keypunch machine functional diagram.

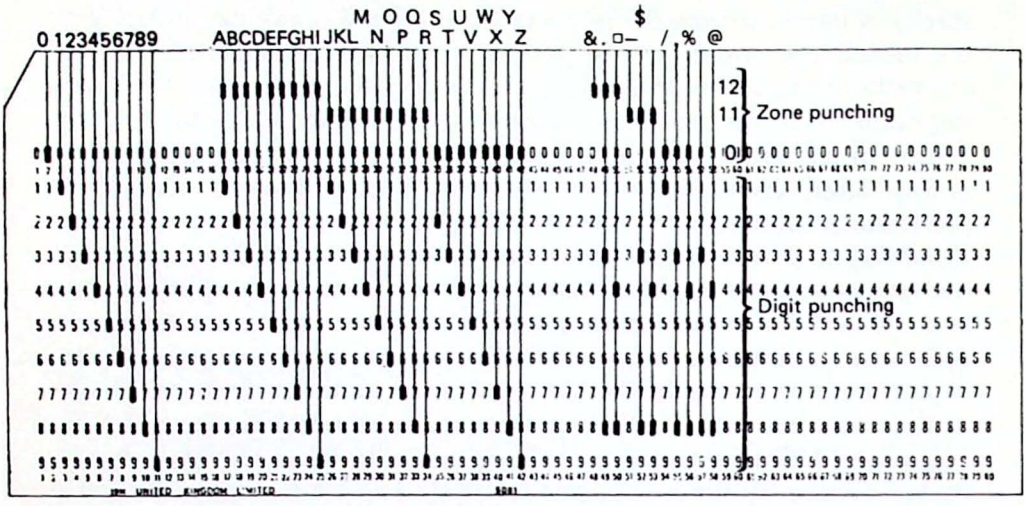


FIGURE 1-2. Hollerith card.

cards are placed in the hopper above and at the right top of the machine. An individual card is fed into the card station (called the *punch station*) on the right center by the operator's pressing a button. The actual punching is performed by pressing keys on a typewriter-like keyboard. Card columns—1, followed by 2, followed by 3, etc. as far as column 80—are punched in sequence with a Hollerith code. The station at the center (called the *read station*) reads cards after they have been punched and is used primarily to duplicate cards. By pressing other buttons the operator collects punched cards in the hopper at the top left of the keypunch.

Figure 1-2 illustrates a Hollerith card punched with appropriate codes. Note the alphanumeric listing on the card top to assist card checking. Other card-handling machines provide complete mechanical data processing for business. Table 1-1 lists some of these with their general

TABLE 1-1. Other card-handling equipment—general functions

Name	*Function*
1. Verifier	Verifies the correctness of a punched card.
2. Sorter	Sorts on selected data fields, and sorts cards into selected stacks.
3. Collator	A multiprocess device that merges card files, compiles, or matches separate card files. It selects records based on contained data. It checks a file sequence.
4. Calculator	Performs simple arithmetic operations (adds, subtracts, multiplies, divides) on groups of cards.
5. Accounting machine	Prepares printed reports.
6. Reproducing punch	Prepares punched-card data records.

functions. Although mechanical methods had lost their impact by 1978, the keypunch is still a most important tool for programmers. Its coding form still reflects Hollerith code formats, and many programs are inputted to computers in this manner. In some instances cards are completely bypassed by using paper tapes, key-to-tape, and key-to-disk equipment. Since this book is not concerned with mechanical data processing, the important equipment of this general type is listed only in Table 1-1. Most of this form of data processing gear is being replaced by computers of various sizes and types. The efficiency, speed, and capacity of electronic computers to handle a sizeable computing load are much greater.

1-4 COMPUTER DEVELOPMENT BEFORE AND DURING WORLD WAR II

1-4.1 Analog Scientific Calculations

Shortly before World War II, Dr. Vannevar Bush and others recognized the need for calculating machines to compute rapidly the responses of complex physical systems. In many branches of engineering, physics, etc., the behavioral response of these systems can be characterized as differential equations. In many cases, the solution form is well-known, but there are numerous coefficient values to be determined before the precise solution is obtained. In other cases, the system is represented by families of equations, each one interacting on another. In either instance, *mechanical differential analyzers* proved themselves handy tools to characterize solutions. Several *mechanical differential analyzers* (MDAs) were constructed between 1928 and 1942. After World War II, the *electronic differential analyzer* (EDA) performed similar computations. Both of these computing machine forms are classified as *analog machines*, because numerical quantities were represented in magnitude as a physical quantity, such as a voltage or the relative angular positions of shafts. For example, one volt might equal 10 miles per hour, whereas 5.7 volts would represent 57 miles per hour. The basic computing element of the *mechanical differential analyzer* (MDA) was the *ball* and *disk integrator* (an invention of Lord Kelvin over 70 years before). Special amplifiers with capacitor feedback performed the corresponding integrator action in the *electronic differential analyzers* (EDA). In addition, the EDAs were much easier to set up for a new problem than the MDAs. The setup of a problem for solution on an *MDA* involved positioning long mechanical shafts and changing gear boxes. In the EDAs, the setup changes involved moving wires around on plug boards (much in the way that a telephone operator makes manual connections), and/or adjusting potentiometer settings. Further advances in the differential analyzer concept was made

in the early 1950s, by the introduction of the *digital differential analyzer* (DDA). This version permitted data quantities to be represented by pulse trains, but still, like its predecessors, was limited to solving problems where differential equations appeared.

Other prominent uses of *analog computers* at the time of World War II involved positioning guns so that the shells would strike the target even though the target was moving rapidly. In general, the gun barrel must be positioned in advance of the target (leading) to allow time for the shell to traverse the distance to the target once it leaves the barrel. Because the necessary "lead" varies with range, angle, the speed at which the target moves, gravity, wind, and initial muzzle velocity, computer solutions to such fire-control problems received much attention by the military services. Analog computers, however, have an accuracy determined by the precision of the analog components that rarely exceeds 5 percent.

1-4.2 Digital Scientific Calculations

Dr. H. Aiken of Harvard University and his staff in the mid-1930s recognized the need for a powerful numerical calculator that was more versatile than the developing MDA class. As a result, Harvard and IBM teamed to design and build the MARK I. This was a true digital machine in that data quantities were represented as *decimal numbers*, not analog quantities. This machine was in operation in 1939, and was followed by other Harvard machines in later years. The mechanization was electromechanical; that is, it included relays, ratchet wheels, etc. Also, the Bell Telephone Laboratories at about the same time produced several of their *relay computers*, employing teletype transmitter/receivers as communication links between the computer and the human world.

The first all-electronic machine (i.e., using vacuum tubes) was the ENIAC installed at the Ballistic Research Laboratories (BRL) at Aberdeen Proving Grounds, Maryland. This machine, designed at the University of Pennsylvania, was much faster than the Harvard MARK I or the Bell machines. However, its organization reflected the special job that it was designed to do, and was not extremely efficient for other scientific computations. Its limitations were quickly recognized, and shortly after ENIAC's installation, EDVAC (a true binary computer) was designed by the staff at the Moore School of the University of Pennsylvania for BRL. Whereas, the ENIAC performed its computing functions by counting pulse trains representing data quantities, the EDVAC performed arithmetic with numbers coded in binary code form. Thus, EDVAC's organization reflected more accurately the future trend in computer design than did ENIAC. The year of 1950 saw not only EDVAC become opera-

tional, but many other computers about the United States and in England, including SEAC, Whirlwind, MANIAC, etc.

The computer explosion that followed was phenomenal. It was an era when the need for computation machines was widely recognized and the basic technology had finally become available for such equipment designs. First, the reliable relay, then the vacuum tube, followed quickly by the semiconductor diodes and transistors led to less expensive and faster forms of logic. The magnetic drum memories and the delay line memories were followed rapidly by the electrostatic memory and finally the magnetic core memories. Even input/output devices improved quickly. Whereas the teletype (at 20 to 30 bits per second) loaded and unloaded the early machines, IBM and others were experimenting with high-speed paper tapes and punched cards for loading and unloading data from a machine at much faster rates. Potter and Shepard had invented different forms of the high-speed paper printers. Work was progressing on readapting the magnetic coated drum as a *file storage* device, rather than as an internal memory. The drum and its descendant the *disk file* have continued on as the chief form of external data file storage. Other magnetic mass storage configuration ideas, such as the National Bureau of Standards slotted disk and the BRC card shuffler, were not totally developed because the inherent mechanical simplicity of rotational drum recording discouraged competition of more sophisticated storage devices. Magnetic tapes, on the other hand, are very effective for file storage, where transactions can occur in a planned sequence over a period of time.

1-5 DATA PROCESSORS AND SOCIETY

1-5.1 General

The U. S. government and/or large universities financed most of the early digital processors. As a result, their early use was widespread in the scientific fields. ENIAC, EDVAC, and SEAC, all government-sponsored and university-constructed, have been mentioned. The first computer designed and built by a commercial organization was UNIVAC I. This machine assisted with the census of 1950, and demonstrated to the public how large computers might be employed in practice for large-scale computation programs.

The application of data processors to the handling of business-type data followed swiftly the early scientific applications. The census of 1950 was digested by a UNIVAC I computer. This, of course, gave wide press exposure to the computerized data reduction process. In 1952, the General Electric Company installed a computer to handle payroll operations

in its Owensboro, Kentucky plant. From this point on, application rapidly followed application. Corporations felt impelled to experiment with computer data processing. Is it a toy—or does it have a useful, potential future that we should not allow our competitor to use first? To try it out appeared to offer the quickest evaluation as to a data processor's usefulness. Consequently, many organizations took the plunge without knowing what to expect. Overenthusiastic entrepreneurs tended to oversell the computer's capability and undersell the adverse aspects of system's planning and required programming for new applications. As a result, many early installations were disappointing experiments.

In spite of the problems, a processing approach developed which handled most of the data processing needs of the '50s and '60s. This technique, termed *batch processing*, required data to be brought to the data center, where it was punched on cards and these were read into the processor. The desired program, usually stored on magnetic tape, was also selected and entered from the tape into the computer memory. The program acted on the data, and the results of the processing were outputted via the printer, or perhaps stored on a second card deck for further processing. The user of the data came to the data center to pick up the results. This form of processing procedure worked well enough for applications that occurred at periodic times, for example, a biweekly payroll preparation or end of the month business accounting.

1-5.2 Control Processing

A completely different form of processing frequently occurs in control applications. For example, suppose that one is planning an on-board aircraft computer system that will perform navigation. Input data to such a system are from the gyrocompass, the indicator of air speed, sensor of drift, altimeter, etc. Although the above-mentioned instruments do not in themselves furnish directly a usable input to a digital computer, devices called *converters* can "convert" the instrument sensor output to a digital form of data input that the digital computer can interpret. Thus, information data are available to the computer, and they are in a digital format that can be digested by the computer. The computer, however, must periodically (probably at least once a second) sample the sensor devices via its converters to upgrade the computations of such important factors as longitude and latitude, ground speed bearing, and distance (or time) to a chosen destination. This computing process differs radically from the *batch process* in that data are supplied continuously and no operator is required to enter a batch of punched cards. The computation process and data output are also continuous. The navigation information may be displayed to the pilot, and/or it may be reconverted to an analog form necessary to automatically operate the autopilot. This

example is a form of real-time processing, where the computations must keep up with the process in the time frame of its natural occurrence. Although the navigation problem was cited, an example could just as well have been an industrial process, such as the computer control of electrical power generation, or the automatic operation of a chemical plant or oil refinery.

1-5.3 Time Share Processing

In the early 1950s, it occurred to staff members at some of the influential universities, namely, Dartmouth and the Massachusetts Institute of Technology (MIT), that a computer could be a useful tool for researchers whose problems were too small to rate a formal batch operation. Teletype terminals were scattered about the campus, which permitted the computer to be programmable and data to be entered from each input program individually and separately. The computer would process each one in turn and return the answer to the appropriate operator's console as a printout. A special operator's programming language—(BASIC)—was developed to make it easy for a user knowing little of formal programming to enter his problem for computer solution on the system. From these primitive experiments, the technique of time sharing has expanded in many directions. For example, one form is the vast airline reservation system, where a customer calls in via telephone to reserve space. The local ticket agent can dial the central computer—in another city, probably—and enter his request via a special keyboard, and the computer will acknowledge and record the transaction. Likewise, reservations can be cancelled or information requested of early available flights having the desired accommodations. Another name for this particular system is *on-line system.*

Other forms of time sharing involve expansions of the Dartmouth and MIT idea to allow computer access via teletype terminals from many miles away—literally across the country. Thus, it is possible to purchase a computing service, prestore your own program, and repeatedly enter new data and shortly read out (via the teletype printout) the computed results. For many applications, this form of operation is highly useful. Large corporations, such as IBM, General Electric, and Honeywell, have time-share networks usable by their widely scattered company offices and plants throughout the country.

Because the rental of telephone lines can be relatively expensive, a growing trend is to install a smaller computer at each plant and permit local access from terminals via internal dedicated facilities. Thus, the necessity to curb excessive communication expenses has restored a trend that is back toward the original Dartmouth and MIT experimental systems.

However, sometimes it is desired to communicate with a fast, large, central processor. Thus, the concept of "remote batching" was conceived. In this approach, the data are transmitted and received via a terminal installed at the user's location at high data bit rates (2400 to 9800 bits/second). Thus, the communication link is used at high rates more efficiently, and the cost to transfer data (on a bits per dollar basis) is much improved. Such a system is helpful if:

1. Batch processing is adequate.
2. One's own computer is overloaded with work.
3. The volume of work is sufficient to warrant purchase of the service, and rental of the necessary terminal equipment.

1-5.4 Communications Processing

For many years, the military services, governments, and businesses have conducted business and forwarded messages worldwide using teletype equipment. This equipment has speed limitations. The ASR-35 units, which are widely used, transmit and receive at 10 characters per second. Sometimes this is intolerably slow in the modern world environment. As a consequence, much work is being done to combine several individual slow-speed teletype messages into a computer word format and transmitting this word at much higher rates from one computer to a second, where it is disassembled and retransmitted on individual teletypes. Thus, the wire link is used much more efficiently. Computers are employed at both ends to assemble and receive the individual message characters. In addition, the computers can store messages temporarily if the transmission system becomes overloaded. When it is possible to resume transmission, messages are recovered from storage and then forwarded automatically to their destination. This general form of data system is characteristically termed a store-and-forward system. The Defense Communication Agency AUTODIN system is an outstanding example.

Presently, the U. S. Government Advance Research Projects Agency (ARPA) is experimenting with the linking together of several university computers from Massachusetts to California. At each computer site, a separate control called the information message processor (IMP), is employed to standardize the word formats from diverse computers, making them all suitable for transmission over the data link in a standard format. In other words, the control processor at the sending data processor end accepts words from the data processor and puts them into the standard form to be sent over the 50,000 bit/second transmission path. At the recovery end, the control processor IMP here con-

verts the transmission path word to a word form to which the receiving data processor can respond. Figure 1-3 outlines the essence of the system. In any case, the computer input/output is linked to the IMP; the data transmitted now are not necessarily TTY data (multiplexed or otherwise).

1-5.5 Specialized Processing

There is still a market for specialized processors. For example, the University of Illinois has constructed Illiac IV, which solves special problems where the input data fit its peculiar organization. In this case, extremely fast processing is obtained by employing banks of individual processors in parallel, all controlled by the same command from a master control processor. This form of computer "system" is termed an *array processor*.

Other problems that seem to lend themselves to special processors or organizations include the problem of controlling aircraft about a terminal airport. The computation rates associated with this general problem, if more than a few aircraft are involved, are so great that the usual organization just cannot handle them on a real-time basis. In this instance, another array organization, termed an *associative array*, has been proposed as a solution.

1-5.6 Minicomputers

The minicomputer is a class of inexpensive versatile machines introduced originally about 1969. They are generally simple to connect to systems and to program. In a sense, every laboratory or every classroom can have its own computer. In this way, many of the costs of contracting with a time-share service can be avoided. A typical minicomputer organization follows that of a higher-performance, more expensive control processor for which a complete set of operating software was already developed and proven. In addition, a complete set of peripherals already were interfaced with the control processor and, hence, are already available to the minicomputer.

Up until 1972 control applications in their various forms dominated the use of minicomputers. With the advent of the microcomputer in this year many control applications, not really complex enough for a minicomputer, started planning to use a microcomputer instead. By 1978 most purely control applications required only microcomputers, except possibly those systems where individual microcomputers were poled by a central computer, most often a minicomputer. Thus, to keep the minicomputer useful, new horizons had to be found for further uses.

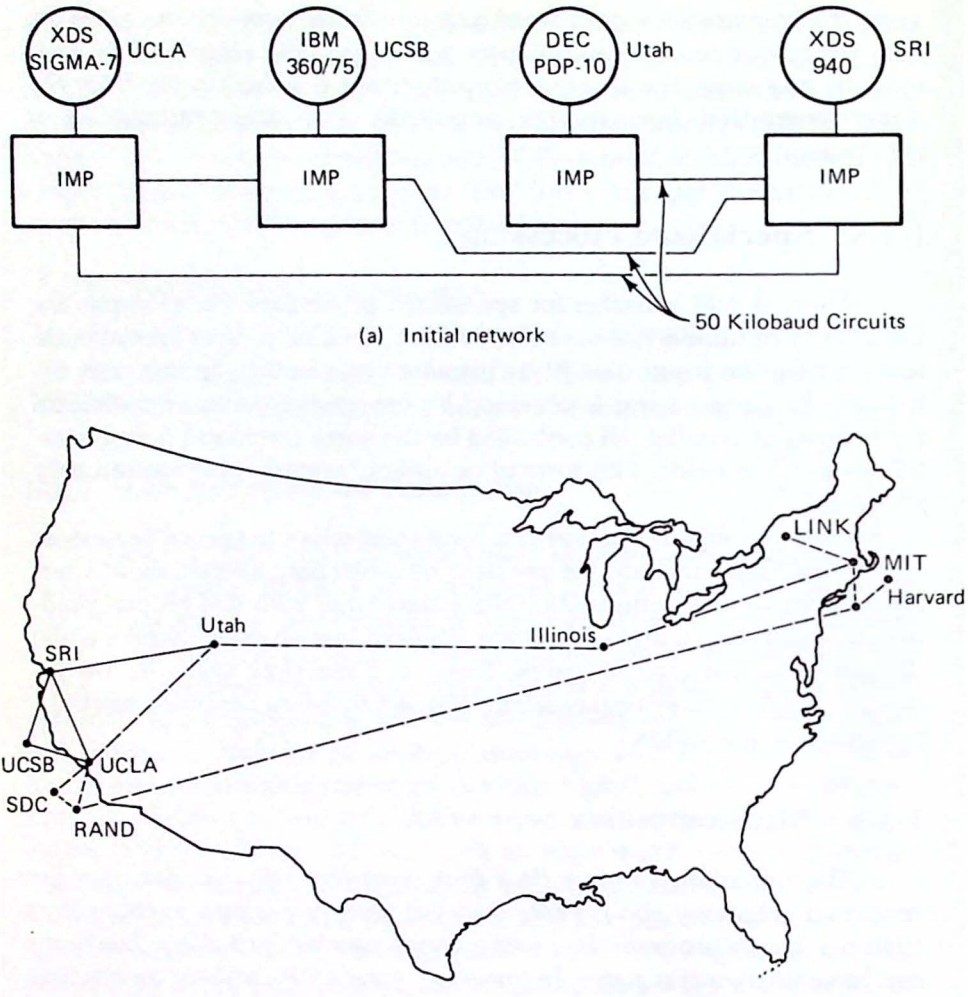


FIGURE 1-3. The form of the ARPA network.

This was not a difficult proposition, because by 1973 minicomputers were as powerful as the large machines of a decade earlier. For example, as will be seen in Chapter 11, the Interdata machines use IBM 360/370 series types of instructions, and this series is the most heavily used in the world today for large-scale operations, including both business and remote time-sharing applications. Thus, minicomputers in 1978 can easily adapt to large-scale operations, and the tendency now is to use minicomputers, not only for control, but business applications as well. This is not to say that all large-scale machines should be scrapped. There are many organizations whose computing facility is a central one (batch

processing) and for most uses such as payroll, stock inventory, accounting, etc. the central facility is very much integrated with the remainder of the corporation's operations. There is little justification to change over, if this system does its job to the satisfaction of management.

1-6 SUMMARY

This chapter is pure introduction. Most topics mentioned are discussed in more detail later in the book. The chapter starts by mentioning early computing devices, such as the Abacus, Napier's bones, Pascal's adding machine, and Babbage's work. Since punched cards were a major means of recording business transactions before 1950, and because keypunch machines are very actively used at most modern data processing establishments, the punch card era is discussed in Section 1-3. The real computer revolution began shortly before World War II. The Harvard machines, the Bell relay computers, the ENIAC, and others, such as EDVAC, SEAC, ORDVAC, WHIRLWIND, etc., proved that the concept and value of high-speed data processing equipment were worth experimenting with. Although immediately following World War II large analog computers flourished, their growth and widespread adoption were limited by their inflexibility and their inherent accuracy limitations. The programmable digital machine, however, can be adopted to an almost unlimited number of applications. Its functioning is completely controlled by internal stored programs that are relatively easily changed. Major system types include batch processing, control, and time-sharing. In addition, the late 1960s saw the birth of the small, inexpensive, easy-to-apply miniprocessor. The cost of these machines is a few hundred to a few thousand dollars; hence, it is not unusual to find computers in small scientific labs, or as part of the actual test experiment instrumentation. In 1974 the microcomputer development started and took over simple computer tasks quickly. By 1978 the minicomputer and microcomputer had merged and were using a microprocessor as the processing element. This development will continue.

1-7 PROBLEMS

1. What is an Abacus?
2. Can you name the contributions of Napier, Pascal, VonLeibniz, Babbage, and Hollerith?
3. a. Why have computers replaced much of the card-handling equipment developed early in this century?

b. Keypunches are still in wide use—where and for what reason?

4. What types of problems are solved by differential analyzers?
5. What are some of the special devices added to computers?
6. What is batch processing?
7. What is control processing?
8. What is time share processing?
9. What is time-share batch processing?
10. a. What is a minicomputer?

    b. Why were minicomputers not developed until the late 1960s?

2

Fundamentals

2-1 INTRODUCTION

This chapter reviews the fundamentals of numbers as applied to computers and some of the codes employed within the systems. It does not treat in depth the topics of conversion of numbers from base to base, or arithmetic in the various base systems. Although these are interesting topics in themselves, it is not necessary to be completely conversant with their details for the material to follow. However, examples are given of binary to octal, decimal-binary conversions, which occur frequently enough in practice to be worthy of special attention.

2-2 NUMBER SYSTEMS

2-2.1 Representation

The way in which numbers are represented within different computers varies. In fact, within the same computer numbers can be represented in different ways, the only thing being common is the fact that number values are represented by groups of 1's and 0's. The most common manner in which a number may be represented in practice is by magnitude. For example, the number 327 in the decimal system is recognized to be $3 \times 100 + 2 \times 10 + 7$ or, in a slightly different format, to be $3 \times (10)^2 + 2 \times (10)^1 + 7 \times (10)^0$.

In the general case, a decimal number can be defined as:

$$\alpha_{n+1} \times {}^{10}n. \ . \ . \ . + \alpha_3 \times 10^2 + \alpha_2 \times 10^1 + \alpha_1 \times 10^0 + b_1 \times 10^{-1} + b_2 \times 10^{-2} + . \ . \ .$$

The α and b terms are called *coefficients* and can be any value from 0 to 9 in the decimal system. The symbol n assumes a value of the highest power of 10 required to represent the magnitude. The number 10 is the *radix* (or base) for the decimal system.

The same series form can be employed to represent any whole or fractional number system based upon Arabic numbers. This general series can be defined as follows:

$$\alpha_{n+1} \times r^{n} \ldots \alpha_3 \times r^2 + \alpha_2 \times r^1 + \alpha_1 \times r^0 + b_1 \times r^{-1} + b_2 \times r^{-2} + \ldots$$

The α terms are coefficients for *whole* numbers. The b terms are coefficients for *fractional* numbers.

The characteristics of the more common number systems employed with computers are characterized in Table 2-1.

Table 2-1.

System	*Radix*	*Coefficient Values*
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Binary	2	0, 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Hexidecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

There is no law that says only these number systems can be employed. It is just that each of these has a reason for being more useful than, say, base 3, base 5, base 9, etc.

The decimal system is useful because it is the number system most familiar to members of a western society using the equipment. The binary system is useful because circuits that recognize a 1 and 0 are easily provided—much more easily than one that has to recognize reliably which is the coefficient of a decimal number by a nine-level voltage. Thus, the ideal number system for the operator is the decimal system, whereas the computer hardware operates more reliably with numbers in binary form. The octal system is something of a compromise. The coefficients 0 thru 7 can be read easily, and the relative magnitude of the number is readily apparent. Also, the octal and binary numbers are directly convertible; thus, it is easy to convert from an exterior-world octal number to an in-the-computer binary number, or vice versa. Such a direct translation between decimal and binary numbers is not achieved nearly so easily.

Another point in favor of octal is the efficiency in the use of the bits for coding octal. All eight octal coefficients can be coded with three binary bits. For decimal coding of coefficients, four bits are required. Also, six-bit code combinations are essentially unused (10–15). In hexidecimal coding, all 16 combinations of four bits are employed. Thus, hexidecimal is as efficient as octal, and some manufacturers (for example, IBM) prefer, at times, to code in this number system. Note decimal BCD is a subset of the hexidecimal code.

2-2.2 Binary

Binary computers normally have the binary point at one end of the binary word. The left-hand end—usually a (1) or (0) to the left of the binary point—will then indicate the sign (− or +) of the number. The reason for the restricted placement of the binary point originates from the less complex logic and/or less complex programs for binary-decimal or decimal-binary conversions. If the binary point is allowed to be placed anywhere, it usually requires two separate conversion techniques—one for the left-hand side of the binary point and another for the right-hand side.

Table 2-2 shows ways in which fractional numbers can be coded in a computer word. The program must remember what each code represents, i.e., binary, decimal, octal, or what? If the program misinterprets, that is, treats a binary number code as decimal BCD or vice versa, errors in the solutions, data manipulations, etc., of course, will occur, leading to chaos—i.e., a faulty solution, if any.

TABLE 2-2 Number Codes

+ Binary	$+20_{(10)} = +20_{(2)}$	0.00000000010100
+ Octal	$+20_{(10)} = +24_{(8)}$	0.000▲000▲000▲010▲100
+ BCD	$+20_{(10)} = +20_{(BCD)}$	0000▲0000▲0010▲0000 (no bit for sign shown)
+ ASC 11	$+20_{(10)} = +20_{(ASC11)}$	0.0100010▲0.01000000
+ Hexidemical (HD)	$+20_{(10)} = +14_{(16)}$	0.00010▲100
− Binary	$-20_{(10)} = -20_{(2)}$	1.111111111101100
− Octal	$-20_{(10)} = -24_{(8)}$	1.111▲111▲111▲101▲100
− Hexidecimal (HD)	$-20_{(10)} = -14_{(16)}$	1.11101▲011

Fractional representations do not satisfy all diversified number operations for digital computers. Some cases where a different form of number representation is useful include:

1. Cases where numbers may differ radically in magnitude (for example, 0.0000013 and 13,730,180).
2. Cases where *imaginary* numbers are to be portrayed.
3. Cases where, in a relatively few instances, a much higher precision is required of the number than normally available in the standard fractional notation.

It is noted that *floating point notation* and *double precision notation* are relatively common methods to avoid these restrictions, but this subject is

deferred to Secs. 5-4.3 and 5-4.4. Not all computers possess the necessary hardware or software to operate in floating point or double precision modes, however.

2-3 CONVERSION—BASE TO BASE

The conversion of a number from one base system to another involves equating the magnitude (sum of the two respective series), or in symbolic form,

Base r_1 series	Base r_2 series
$=n$	$=m$
$\Sigma \alpha_n r_1^n + \ldots r^0$	$\Sigma\, b_m r_2^m + \ldots b_0 r_2^0$
0	0

As an example, consider computing 428 in base $10(428_{(10)})$ to base 4 $(X_{(4)})$. The radix powers in base 4 are

$$4^0 = 1$$
$$4^1 = 4$$
$$4^2 = 16$$
$$4^3 = 64$$
$$4^4 = 256$$
$$4^5 = 1024$$
$$\text{etc.}$$

The coefficients in base 4 can only be 0, 1, 2, and 3. Since the total value of 428 is less than $1024 = 4^5$, the largest coefficient term will be associated with 4^4, since 4^4 goes into 428 one time. Therefore, the coefficient of the desired 4^4 term is 1. There is left $428 - 256 = 172$. The coefficient of the 4^3 term is 2 (since $2 \times 64 = 128$). Thus, the remaining quantity is $172 - 128 = 44$. The 4^2 term will be $2 \times 4^2 = 32$. The remainder is $44 - 32 = 12$. The 4^1 term multiplied by $3 = 12$, which is the remainder of the series required to represent the total 428 in base 4. Thus, $428_{(10)} = (1 \times 4^4) + (2 \times 4^3) + (2 \times 4^2) + (3 \times 4^1) + (0 \times 4^0)$ or $428_{(10)} = 12230_{(4)}$.

The usual way to convert from a *whole* decimal number to one of smaller radix is as follows:

1. Divide the number by the new radix continuously, noting the remainders, until the quotient is zero.
2. Arrange the remainders with the first determined as the least

significant digit (LSD) and the last determined as the most significant digit (MSD).

To convert from a *fractional* decimal number to a smaller fractional radix:

1. Multiply by the new radix continuously (noting the left-hand overflow beyond the decimal point) until the product is zero. This overflow is the desired digit in the new base.
2. Arrange the overflow with the first one as the MSD and the last as the LSD.
3. The process can be stopped when the desired accuracy is achieved, since the product may not go to zero.

EXAMPLE:

$125_{(10)}$ equals?$_{(2)}$

This process is for a whole number. Thus, according to the first method above, dividing successively by the radix and noting the remainder in each case, we have

$$\begin{aligned} 125/2 &= +\ 62 + 1 \\ 62/2 &= +\ 31 + 0 \\ 15/2 &= +\ 7 + 1 \\ 7/2 &= +\ 3 + 1 \\ 3/2 &= +\ 1 + 1 \\ 1/2 &= +\ 0 + 1 \end{aligned}$$

Therefore, the binary equivalent is

$$111101_{(2)}.$$

EXAMPLE:

$469_{(10)}$ equals?$_{(4)}$

The solution follows the previous example, except for the change in radix to base 4.

$$\begin{aligned} 469/4 &= 117 + 1 \\ 117/4 &= 29 + 1 \\ 29/4 &= 7 + 1 \\ 7/4 &= 1 + 3 \\ 1/4 &= 0 + 1 \end{aligned}$$

The final result is then

$$13111_{(4)}.$$

EXAMPLE:

$0.523_{(10)}$ equals $?_{(8)}$

Since this is a fraction, the second method is applicable. Multiplying by the radix and noting the remainder to the left of the decimal point, we have $0.523 \times 8 = 4.184$, $0.184 \times 8 = 1.472$, $0.472 \times 8 = 3.776$, *and* $0.776 \times 8 = 6.208$, *etc* $= 0.4136_{(8)}$ *after arranging the digits to the left of each decimal point. The process is obviously not going to zero; hence, it will not be continued. Employing the overflow digit to the decimal points left, and placing it within the series, we finally have* $4 \times 1/8 = 0.5$, $1 \times 1/64 = 0.0156$, $3 \times 1/512 = 0.00586_{(10)}$ *and* $4 \times 1/4096 = 0.00098$. *Thus, the fraction* $0.4136_{(8)}$ *sums to* $0.52244_{(10)}$, *which is a close approximation to the necessary results. Further steps would close the gap more closely.*

Since this text is not primarily concerned with the general aspects of number base conversion, it is assumed at this point, if one had to convert between two bases, that he could convert from the first base to base 10 and from base 10 to the second base.

There is one notable exception, however. Conversion from base 2 to base 8, and vice versa, can be accomplished by inspection.

For example, if the number in base 2 is

$$101_{\blacktriangle}\ 001_{\blacktriangle}\ 010_{\blacktriangle}\ 111$$

it can be written in base 8 by sectoring in groups of three starting at the decimal (octal) point. Thus, the octal equivalent is read via the sectored codes, or $5,127_{(8)}$.

To go from octal to base 2, the process is reversed, i.e., the octal number is converted to the equivalent binary code for each digit. Thus,

$213765_{(8)}$
becomes
010 ▲ 001 ▲ 011 ▲ 111 ▲ 110 ▲ 101

Note: ▲ indicates sectioning of the binary number into 3-digit groups.

2-4 CODED NUMBERS (BCD)

Because the external world is a decimal world, it frequently becomes necessary to convert from decimal numbers to binary when one is going from an external device (for example, a keyboard) into the computer.

The decimal inputting process runs something like this. An operator types on a keyboard 57321, etc. The decimal values are represented by individual keys. These keys put codes such as those shown in Table 2-3 into the computer.

TABLE 2-3 Codes Stored as BCD

Number Pressed	*Number Stored*	
1	1	0001
2	20	0010
3	300	0011
7	7 000	0111
5	50 000	0101

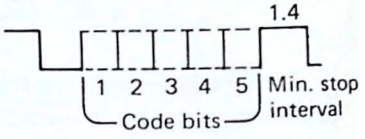
2-5 CODED NUMBERS AND LETTERS

Four-bit codes will represent all numbers of base 10, with six codes left over. In order to code letters as well, codes longer than four bits are necessary. As a minimum number of bits it generally required six bits (64 combinations) to code both numbers and the alphabet and some punctuation. In the past, there have been some codes that worked around this limitation. Two examples widely known are the five-bit Baudat code employed for Teletype and the three-punch Hollerith codes used for

Baudot Five-bit Code							
Character		Impulse Position					
Uppercase	Numbers/ Punctuation	1	2	3	4	5	
A	-	•	•				
B	?	•			•	•	
C	:		•	•	•		
D	$	•			•		
E	3	•					
F	!	•		•	•		
G	&		•		•	•	
H	#			•		•	
I	8		•	•			
J	'	•	•		•		
K	(	•	•	•	•		
L	)		•			•	
M	.			•	•	•	
N	,			•	•		
O	9				•	•	
P	0		•	•		•	
Q	1	•	•	•		•	
R	4		•		•		
S	BELL	•		•			
T	5					•	
U	7	•	•	•			
V	;		•	•	•	•	
W	2	•	•			•	
X	/	•			•	•	
Y	6	•		•		•	
Z	"	•				•	
LETTER(shift to lowercase)		•	•	•	•	•	
LETTER(shift to uppercase)		•	•		•	•	
SPACE				•			
CARRIAGE RETURN					•		
LINE FEED			•				
BLANK							

Presence of • indicates marking impulse

Absence of • indicates spacing impulse



There is also an eight-bit ASC11 code. See Table 2-4.

FIGURE 2-1. Baudot teletype code.

punched cards. The five-bit Baudat code employed a shift key to allow the five-bit code to be interpreted in $2 \times 2^5 = 64$ code combinations. The Hollerith code of Chapter 1 employed up to three punched holes. The Baudat code is illustrated by Fig. 1-2 and 2-1.

A binary equivalent of decimal numbers, usually termed binary-coded-decimal or simply BCD, is widely employed. Its most usual form is

1	0001
2	0010
3	0011
4	0100

5	0101
6	0110
7	0111
8	1000
9	1001
0	0000

Note that the codes stored in the computer might look like the following:

0101 1001 . . . 0111 0010 0001

This is *not binary*, however. It is BCD, and to convert to binary requires some additional processing, usually accomplished by a simple program, for example:

$$1\times 10^{0}=1$$
$$2\times 10^{1}=20$$
$$7\times 10^{2}=700$$
$$9\times 10^{n-1}=90\ldots000$$
$$5\times 10^{n}=500\ldots000$$

The total stored is the sum $= 1+ 20+ 700 \ldots + 9\times 10^{n-1}+ 5\times 10^{n}$

It is also common practice to observe the reverse process—that is, to display a binary number in its decimal equivalent form. The computer program computes BCD codes from the binary code, and these BCD codes are outputted to a display device as a number display or a printer.

Table 2-4 represents a typical six-bit code employed to transmit alphanumeric data alone. However, to gain further versatility, the trend is to seven and eight-bit codes, generally known as ASCII or USASCII.

TABLE 2-4 Alphanumeric six-bit codes

1.	000000	A.	110001	J.	100001	S.	010010
2.	000010	B.	110010	K.	100010	T.	010011
3.	000011	C.	110011	L.	100011	U.	010100
4.	000100	D.	110100	M.	100100	V.	010101
5.	000101	E.	110101	N.	100101	W.	010110
6.	000110	F.	110110	O.	100110	X.	010111
7.	000111	G.	110111	P.	100111	Y.	011000
8.	001000	H.	111000	Q.	101000	Z.	011001
9.	001001	I.	111001	R.	101001		
0.	001010						

These codes, in addition to transmitting alphanumeric codes and punctuation, also provide for *control character* transmission, which allows *remote control* systems to be implemented over communication links. Message errors, when detected at the receiver, can result in a retransmission of erroneous data blocks.

A form of USASCII code is illustrated by Table 2-5. Codes of this form are now considered standard for data transmission, via teletype, digital data links, and for interfacing computers via magnetic tapes, printers, paper tapes, data modems, etc.

TABLE 2-5 ASC II codes

b_7					0	0	0	0	1	1	1	1
	b_6				0	0	1	1	0	0	1	1
		b_5			0	1	0	1	0	1	0	1
b_4	b_3	b_2	b_1		*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	0	0	0	0	NUL	DLE	SP	O	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	:
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	∩	n	~
1	1	1	1	15	SI	US	/	?	O	—	o	DEL

ENQ enquiry
ACK acknowledge
BEL bell
BS back space
HT horizontal tab
NUL null
SOH start of heading
STX start of text
ETX end of text
ETB end of transmission block
EOB end of transmission
CAN cancel
EM end of media
SUB substitute
ESC escape
SI shift-in
DLE data link escape
DC1 — DC4 device control
NAC negative acknowledge
SYN synchronize
FS file separater
GS group separater
RS record separater
US unit separater
LF line feed
VT vertical tab
FF form feed
CR carriage return
SO shift out
DEL delete

2-6 BINARY ARITHMETIC

Addition in the binary system is quite simple with the use of the following rules:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$

$1 + 1 = 0$ plus a carry of 1 to next higher column

$1 + 1 + 1 = 1$ plus a carry of 1 to next higher column

EXAMPLE: *Add the following numbers in both decimal and binary form:*

	Binary	Corresponding Decimal
	01101	13
	01010	+ 10
Solution:	$10111_{(2)}$	$23_{(10)}$

Note: In adding the fourth column of the binary example, a carry is generated.

The rules for substraction are

$0 - 0 = 0$
$1 - 0 = 1$
$1 - 1 = 0$
$0 - 1 = 1$ with a borrow of 1 from the next higher column

EXAMPLE: *Subtract the following numbers in both decimal and binary form:*

	Binary	Corresponding Decimal
	01110	14
	−01011	−11
Solution:	+00011	+3

This example requires two borrows. (0 − 1 = 1 + borrow from next left-hand column. When 1 is borrowed, the number becomes 0 and subtracting a 1 makes it 0; therefore, another borrow is needed.)

Note: In the preceding example, it is necessary to borrow for both the second and third columns. In the following example, the necessary borrowings are from the third and fourth columns.

EXAMPLE: *Subtract the following numbers in decimal and binary form:*

	Binary	Corresponding Decimal
	11010	26
	−01100	−12
Solution:	$01110_{(2)}$	$14_{(10)}$

Multiplication in the binary system also follows the same general rules as decimal multiplication. However, learning the binary multiplication table is a trivial task because

$$1 \times 1 = 1$$
$$1 \times 0 = 0$$
$$0 \times 0 = 0$$
$$0 \times 1 = 0$$

It is that simple! You now know the multiplication table for elementary binary multiplication.

EXAMPLE: Multiply the following numbers in decimal and binary form:

Binary	Decimal
01101	13
×01001	×9
01101	$117_{(10)}$ (check)
00000	
001101 add	
00000	
0001101 add	
01101	
01110101 add	
00000	
$001110101_{(2)}$ Final answer	

Note that, whenever there is a 1 in the multiplier, the multiplicand is added to the previous sum but is displaced one place to the left. Whenever 0 is in the multiplier, a string of zeros is added instead. The addition is performed after each partial sum to illustrate one manner in which an elementary digital computer actually performs the steps of multiplication.

In actual practice, whenever a 0 appears in the multiplier, *no addition is required*, only a *left shift*. As a result, the past example may be reduced to

$$\begin{array}{r} 01101 \\ \times\ 01001 \\ \hline 01101 \\ 01101SS \\ \hline 01110101_{(2)} \end{array} \quad \text{(S = left shift)}$$

A computer would also follow this procedure in performing multiplication.

The division process is performed in a manner similar to decimal division. The rules for division are

$$0 \div 1 = 0$$
$$1 \div 1 = 1$$

EXAMPLE: *Divide the following numbers in decimal and binary form:*

Binary		Decimal	
	0111 (quotient)		7 (check)
0111	⌊0110111 (dividend)	7	⌊55
	00111		49
	001101		6 (remainder)
	000111		
	001101		
	000111		
	000110 (remainder)		

Note: When the divisor of the example above is subtracted in the first step from the four most left-hand digits, the result would be negative. Therefore, a 0 is entered in the quotient, and the divisor is shifted one place right. Now a valid subtraction can take place, and a 1 is entered in the next position of the quotient. The divisor is shifted one more place to the right, and a third subtraction is attempted, etc.

2-7 COMPLEMENTS

The true complement of a number is obtained by subtracting each digit of the number from the radix minus 1 and then adding a 1 to the least significant digit. The true complement of a decimal number is known as the *tens* complement. The true complement of a binary number is the *twos* complement.

Examples of true complements are

1. For decimal 57, the complement is

$$100 - 1 \text{ (or } 99) - 57 + 1 = 43$$

The true complement may also be found directly by subtracting

$$100 - 57 = 43$$

2. For the true complement of binary 110111, change all the (1s) to (0s) and (0s) to (1s) and *add* a 1 to the LSD.

$$001000 + 000001 = 001001$$

which is the true complement of 110111.

In equation form, the ten's complement is

$$. . .9999 - S + 1 = 1 \times 10^n - S = \overline{S}_{10}$$

where n is the word length in digits, S is the number whose complement is to be found, and $\overline{S}_{10}$ is the true complement of a decimal number.

The equation for a number which is the nine's complement is

$$. . .9999 - S = 1 \times 10^n - S - 1 = \overline{S}_9$$

where n and S are defined as before and $\overline{S}_9$ is obviously the nine's complement. The nine's complement of the decimal system is an example of a radix-minus-1 complement.

The radix-minus-1 complement is not really of great use with the decimal system, since the computation of the nine's or ten's complement is equally difficult. For the binary system, however, where the radix-minus-1 complement (or one's complement, as it is often called) is obtained merely by reversal of (1s) and (0s), the process simplifies both subtraction and division. A 1 is added to the LSD only if the two's complement is desired; hence, the one's complement is usually one step less complicated to obtain. A simple hardware approach, however, permits the two's complement of a binary number to be generated if the system is serial.

The principal use of a complement number is to perform either subtraction or division by the process of addition. To perform a subtraction, the subtrahend is put in its complement form, and the minuend and the complement subtrahend are added. The result of this addition turns out to be the difference of the two numbers; i.e., a subtraction results.

The reason for performing subtraction by this method is that a computer never needs to be taught to subtract, since subtraction may be performed by addition. Thus, the computer can add, subtract, multiply, and divide by the simple process of addition and shifting only. This is often desirable, since complementation, especially the one's complement, is generally easily performed in a computer and usually saves hardware. The example to follow illustrates complementation with a ten's complement.

EXAMPLE: Subtract 57 from 89 using complementary decimal addition.

Solution: Decimal number normal subtraction

```
  89 (minuend)
 -57 (subtrahend)
  32
```

To find the true complement

```
  100
 - 57
  43 (complement of 57)
```

To perform a subtraction

```
   89
  +43 (complement of 57)
  132 (ignore the 1)
```

The answer is the same as that established by normal subtraction.

The example below illustrates complementation with a nine's complement.

EXAMPLE: Subtract 57 from 89, using the nine's complement.

Solution:

```
  99
 -57
  42  (nine's complement of 57)
```

To perform the subtraction with the nine's complement, add

```
          89
          42 (nine's complement of 57)
        1 31
carry 1   └1
          32
```

This process also gives the original result.

The example following illustrates binary complementation (radix minus 1), or one's complement.

EXAMPLE: Subtract decimal number 10 from 16 in binary, using the one's complement.

Solution:

(minuend)	16	10000	
(subtrahend)	−10	−01010	(for complement, reverse 1s and 0s)

Proceeding in the same manner as in the decimal examples above, we obtain

```
          10000
          10101 (complement)
add     1 00101
carry 1   └──→ 1
          00110 (answer = decimal 6)
```

The basic difference between the two's complement and the one's complement is where and how the 1 is added. In the two's complement, the 1 is added directly to the result *after* the 1s are changed to 0s and the 0s to 1s. In the one's complement, the 1 is added by allowing the carry 1 to be added to the LSD. This is known as *end-around carry*. The last two examples illustrate end-around carry.

2-8 SIGNED BINARY NUMBERS

A binary number representation can be designed to represent both positive and negative numbers. There are two favorite approaches. One is to represent either positive or negative number magnitudes in straight binary code with a "tag" bit (outside the number range) as the sign. Often a 1 in this bit position represents − and a 0 represents +. This is sign-magnitude representation.

The other method of representing binary numbers is to define the positive number in straight binary code (the same as sign-magnitude) but to indicate negative numbers in complement form. Again, the 0 and 1 are assigned to represent + and −, but the determination of the 0 or 1 here is accomplished automatically by the arithmetic operations during the processing itself. Examples of the two representations appear in Table 2-6.

When fixed-point numbers are defined in the pure binary system, it is customary to assign the binary point to the extreme right or extreme left. With the binary point fixed, all numbers represented by the system

will be integers or fractions but not mixed. Binary fractions have the binary point before the most significant bit—usually the extreme left bit, whereas an integer in binary code form has the binary point at the opposite extreme, to the right of the least significant bit. Early serial computers in some cases were integer machines and placed the bit to the right of the LSB. In the EDVAC another bit to the right of the sign bit indicated whether the word was a command or a number. This arrangement was advantageous in early machines, because the number representation was sign-magnitude and the serial numbers were moved LSB first, to the right with the greater-magnitude bits following the LSB in sequence. In this way the sign of the number was known as soon as the transfer began.

In both systems of binary number representation, two's comple-

Table 2-6 Sign-magnitude vs complement number representation

Number value	Sign magnitude*	2s complement
15	0.001111	0.001111
14	0.001110	0.001110
13	0.001101	0.001101
12	0.001100	0.001100
11	0.001011	0.001011
10	0.001010	0.001010
9	0.001001	0.001001
8	0.001000	0.001000
7	0.000111	0.000111
6	0.000110	0.000110
5	0.000101	0.000101
4	0.000100	0.000100
3	0.000011	0.000011
2	0.000010	0.000010
1	0.000001	0.000001
0	0.000000	0.000000
−1	1.000001	1.111111
−2	1.000010	1.111110
−3	1.000011	1.111101
−4	1.000100	1.111100
−5	1.000101	1.111011
•	•	•
•	•	•
−15	1.001111	1.110001
etc.	etc.	etc.

**By the notation employed, the numbers really represented are* n × (2^{-6}), *where* n *is the number value in the left column. xxxxxx represents* 2^{-6}.

ment and sign magnitude a 0 to the left of the binary point represents a positive number. Also a 1 to the left of the binary point represents a negative number. To the right of the binary point the number magnitude is expressed, but in each case by a different method. For sign magnitiude method the magnitude of the number for either positive or negative numbers is expressed by an increasing binary code (straight binary). For two's complement, positive binary numbers are the same as sign magnitude (an increasing binary code), but negative numbers are expressed as the complement code. This is illustrated by Table 2-6.

The later binary computers are primarily two's complement arithmetic machines with the sign bit assigned the extreme left-hand position. The number represented is fractional, and negative numbers are coded in complemen- form. This organization is favored in recent binary machines, because complement arithmetic allows the sign to be computed without delay automatically during additions and subtractions and also without adding additional hardware to the arithmetic section. Multiplication and division require some adjustment to the partial products and quotients generated, but this is not difficult to include. A comparison of addition and subtraction of *sign-magnitude* and *complement* arithmetic is given by Figs. 2-3 and 2-4.

Multiplication and division can be implemented more easily by the *sign-magnitude* approach, since the machine merely observes the respective signs of the multiplier and multiplicand involved and assigns the necessary sign to the product. For the two's *complement* representation it is not uncommon to complement negative numbers and multiply and divide in identical fashion to the *sign-magnitude* counterpart. At the completion of the process, a negative product is obtained by complementing and assigning a 1 to the sign position.

a. Addition of two positive numbers

Decimal	*Binary*
+13	0.001101
+17	0.010001
+30	0.011110

b. Addition of a positive and negative number

Decimal	*Binary*	*Binary*
+13	0.001101	0.001101
−17	−1.010001 2's comp.	0.101111
−4		0.111100 2's comp.

1.000100 Answer

c. Addition of two negative numbers

Decimal	*Binary*	*Binary*
−13	1.001101 2s comp.	0.110011
−17	1.010001 2s comp.	0.101111 2s comp.

1.011110 Answer

a. Addition of positive numbers

Decimal	*Binary*
+13	0.001101
+17	0.010001
+0	0.011110

b. Addition of a positive and negative number

Decimal	*Binary*
+13	0.001101
−17	1.101111
−4	1.111100 (−4 in 2s complement form)

c. Addition of two negative numbers

Decimal	*Binary*
−13	1.110011
−17	1.101111
−30	1.100010 (−30 in 2s complement form)

When one is multiplying two positive numbers, the result is simple; for example,

13	0.001101
×17	×0.010001
91	.001101
13	.001101sss
+221	0.0011011101

Although there are several ways to perform signed multiplication, one of the easiest is to multiply both numbers together, remembering the sign, and to then complement the results, since a positive times a negative number is negative. For example, then

13	0.001101
×−17 (comp)	×0.010001
−221	.001101
	.001101sss
	0.0011011101

Now complement the number to the right of the binary point and the final results are obtained as follows:

1.1100100011

Recomplement to check and we have 0.0011011101—so it does check.

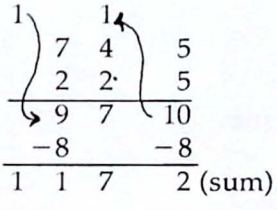
2-9 OCTAL NUMBER SYSTEMS

Octal notation is derived from three-bit binary code. The relationship to the binary code in octal is represented by the following listing. Octal digits include 0, 1, 2, 3, 4, 5, 6, and 7.

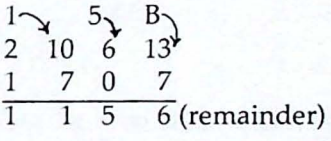
Octal	*Binary*	*Octal*	*Binary*
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Addition and subtraction in octal

Addition



Subtraction



Octal Addition Table

	1	2	3	4	5	6	7
1	2	3	4	5	6	7	10
2	3	4	5	6	7	10	11
3	4	5	6	7	10	11	12
4	5	6	7	10	11	12	13
5	6	7	10	11	12	13	14
6	7	10	11	12	13	14	15
7	10	11	12	13	14	15	16

Octal multiplication table

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	10	12	14	16
3	3	6	11	14	17	22	25
4	4	10	14	20	24	30	34
5	5	12	17	24	31	36	43
6	6	14	22	30	36	44	52
7	7	16	25	34	43	52	61

Multiplication

```
        3 6 5
        2 1 3
    ---------
    1 1 2 7
        2 1
    ---------
    1 3 3 7   ×3
    3 6 5     ×1
    ---------
    5 2 0 7
  7 5 0       ×2
  ---------
  7 2 2 0 7
  1
  ---------
1 0 2 2 0 7
```

2-10 HEXADECIMAL (HD) NUMBER SYSTEMS

Octal system digits using three-bits went from 0–7. HD systems using four-bits go 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

HD	*Binary*	*HD*	*Binary*
0.	0000	9.	1001
1.	0001	A.	1010
2.	0010	B.	1011
3.	0011	C.	1100
4.	0100	D.	1101
5.	0101	E.	1110
6.	0110	F.	1111
7.	0111		
8.	1000		

Hexadecimal addition and subtraction

Addition

```
  F C A 5
  1 A 2 B
  -------
  0 5 C 0
carries 1 1 0 1
  -------
  1 1 6 D 0   (sum)
```

Subtraction

```
  F C A 5
  1 A 2 B
  -------
  E 2 7 A   (remainder)
```

The subtraction problem could have been worked by converting 1 A 2 B to the HD 16 complement and adding. To convert to HD complement, subtract 1 A 2 B from F F F F and add 1 to LSB. This is the same process as for the two's complement process.

```
F F F F
1 A 2 B
-------
E 5 D 4
       1
-------
E 5 D 5  (complement 1A2B)
```

Hexadecimal multiplication

```
  36F      36F      36F      36E      36F
 ×BE2     ×2       ×E       ×B      ×BE2
-----    ----     ----     ----     ----
 6DE      6EE      A42      12A      6EE
   1               25D      249 ---->3012
 6EE              2F12      25BA --->25BA
                     1                ----
                  3012               28C20E
```

HD multiplication table

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	58
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

HD addition table

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	1A	1B	1C	1D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17

HD addition table (cont.)

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

2-11 CONVERSIONS

Hexadecimal to decimal example (E62)

E= 14× 16= 224

Add 6 6

230× 16 3680

Add 2 2

$3682_{(10)}$

Decimal to hexadecimal

	Remainder
16 ⟌ 3682	
16 ⟌ 230	2
14	6

Since 14= E, E therefore, $3682_{(10)}$= E62 $_{(HD)}$

2-12 SUMMARY

This chapter summarizes the topics of number and number conversions as applied to various versions of digital minicomputers. It also outlines come of the more important codes (BCD, octal and hexadecimal) as well as Baudot, USAC11, Hollerith, and 6-bit alphanumeric codes. The chapter concludes with examples of binary, octal, and hexadecimal (HD) arithmetic.

2-13 PROBLEMS

1. $525_{(10)}$= $?_{(4)}$
2. $.625_{(10)}$ = $?_{(5)}$
3. From Fig. 2-3 what are the codes for A, S, 2, etc.?

4. Add+25 to+35 in binary code.
5. Subtract+25 from+35 in binary code. What is the 2's complement of+25?
6. Multiply+8×+15 in binary code.
7. Write Problem 6 in octal and hexadecimal code and show that the results of multiplying are equal.
8. What is the hexadecimal conversion to decimal of F23?
9. Reverse problem 8 to recover F23.
10. Find the hexadecimal true complement of 3D1EC.
11. Find the octal true complement of 47132.

Note: For Problem 10 the complement is found by subtracting each hexadecimal digit from 15. In Problem 11 each octal digit is subtracted from 7. A 1 is then added to the least significant digit in each case to find the true complement.

3

Binary Control and Data

3-1 INTRODUCTION

This chapter discusses in a general way some of the fundamentals of a binary computer's operation and organization. Data in binary systems are a collection of (1) and (0) codes. Different machines, reflecting their individual designs, treat these groupings in different ways. Correct groupings of binary bits represent either data numbers or letter codes or instructions. The chapter concludes with a bare-bones block diagram of a computer and how it is connected to a system.

3-2 DATA IN BINARY SYSTEMS

Data in binary systems exist as (1) and (0) combinations. A "1", for example, could be+5 volts, while a "0" could be ground (0 volts). Other ways may be used to represent binary bits; for example, a pulse occurring or no pulse occurring in a specified time interval, or the existence of a directional magnetic flux in a magnetic core. The coefficient of the binary level term as from Chapter 2 is defined by its position in the code word. In early machines (there were no minicomputers in those days) before 1960, it was customary to transfer data as a single stream most often with the lowest order bit (LSB) leading and higher order bits following in sequence. This ordering resulted primarily from the type of machine in vogue at the time (delay lines or magnetic drums), and the desire to reduce electronic circuitry to a minimum (for example, a single adder was required to serially sum two words). The time to completely transfer a 40-bit word was typically 40 microseconds (μsec), where each bit was clocked at a 1-megahertz rate (1 μsec between bits). For a 40-bit word this required 40 μsec to transfer a word from place to place.

The development of integrated circuits (ICs), medium-scale integrated circuits (MSICs), and large-scale integrated circuits (LSICs) and

the wide adoption of random access memories has popularized parallel word transfers. With the development of all types of the above-mentioned circuits, all-parallel designs are no longer prohibited, and the higher-speed operation is sufficient justification, let alone the lower cost.

A group of bits accessed in parallel at one time from memory is either a word (16 bits, etc.) or a byte (8 bits). The word can be allotted as a number or a computer instruction (sometimes called a command). If it is a number, the entire binary word can be in octal or hexadecimal code. The instruction word [also groups of (1s) and (0s)] has an entirely different function. Its groupings are also subdivided in practice. In use it is transferred to address and operation code decoders where the bits are appropriately interpreted as control information. In Figs. 3-1 and 3-2 are typical examples of how data and command words are organized where the program controls the sequence and interpretation. If the program misinterprets the control or data word, chaos results and the program bombs out. Such catastrophes result from programming errors, hardware failures, etc.

FIGURE 3-1. 16-bit instruction formats.

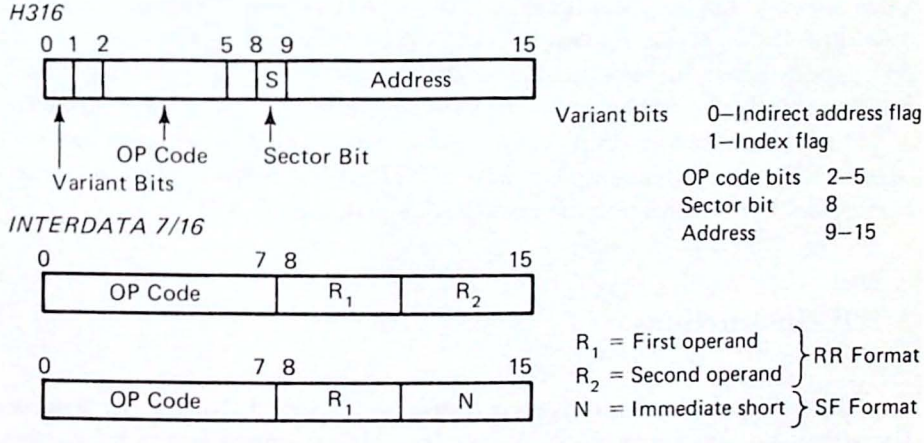
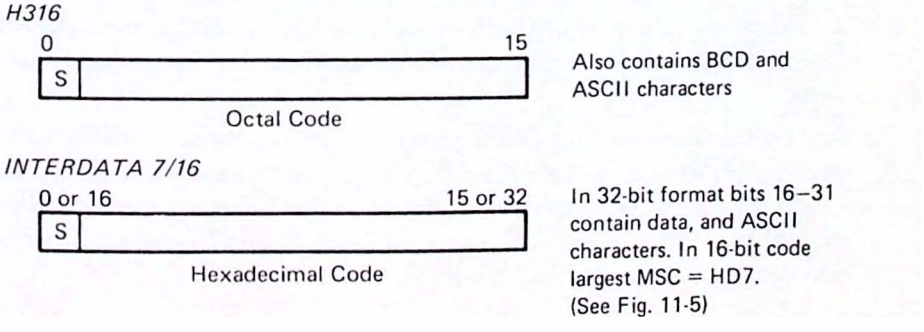


FIGURE 3-2. Examples of data codes.



3-3 PROGRAM COMPONENTS

Programs consist of (1) a charge of instructions, (2) any constants required, and (3) data (words or bytes). If these data are obtained at fixed periods from sources *outside the computer*, the data may be stored in the same manner as *constants*. If, however, the data are obtained from real-time sources continually, as the program progresses, or if the data are the result of other computations, then they are termed *variable* data and are stored in a memory area where they may be modified as the program progresses.

The general operation of a computer requires that the instructions be accessed and executed in sequence. Constants and variable data are called up as needed by the program instructions. The results of computation (also variable data) are restored in memory awaiting further operations. Actually, the three constituent types are stored in the same memory. In any case, the programmer *must* control the proper interpretation of binary words by his program.

In some special-purpose digital computers (usually termed *controllers*) the instruction sequence and constants can be stored in *fixed-memory* areas that are not easily alterable. These memories, termed read-only memories (ROM's), are loaded initially on installation of the system, and no changes are made afterwards to their contents. They can be "read" but not electrically *written* into. For *variable* data, however, the flexibility of both reading and writing is necessary and a random access memory (RAM) is the class to be employed here. Often the term *Scratch pad memory* applies here—especially if the RAM is small and fast.

3-1.1 Instructions

The bit groups within the word will be decoded during the instruction execution processes. For example, bit codes represent such items as:

1. Operation codes. Identify the instruction type.
2. Memory addresses. Memory cells must be addressed during any transfer to or from memory. Each address has a unique binary code.
3. Indexing control (or other means of instruction modification). Index registers can be called up by unique codes which identify the index register to be employed. The bit codes here have been called *variant bits*. Also, a particular *variant bit code* may be employed to identify when indirect addressing is to be performed.

4. Shifting. Bit codes may be employed to identify the number of bits to shift a character, etc. in special instruction types.

Example of instruction types (formats) are shown in Fig. 3-1.

3-3.2 Constant and Variable Data

Data are generally stored in the standard number or character format for the problem at hand and the particular machine. Examples would be pure binary, octal, hexadecimal, and possibly binary coded decimal (BCD) or ASC11 code. Examples are shown in Fig. 3-2.

3-4 COMPUTER COMPONENTS AND SYSTEMS

A block diagram for any typical computer is approximately the same. One has a memory, where data and commands are stored, an arithmetic unit, which performs arithmetic operations, a control section, and an input /output (I /O) which communicates with external devices and systems. Fig. 3-3 shows the block diagram for our basic computer-processor. Likewise, Fig. 3-4 expands the last figure to show typical I /0 functions with a few sample I /0 peripherals attached. The three sections (excluding memory) are generally called a central processing unit (CPU).

3-5 SUMMARY

This chapter examines parallel and serial word transfers and subdividion of binary data words into characters [BCD, octal, ASC11 and hexadecimal (HD)]. It points out that binary words can be instructions,

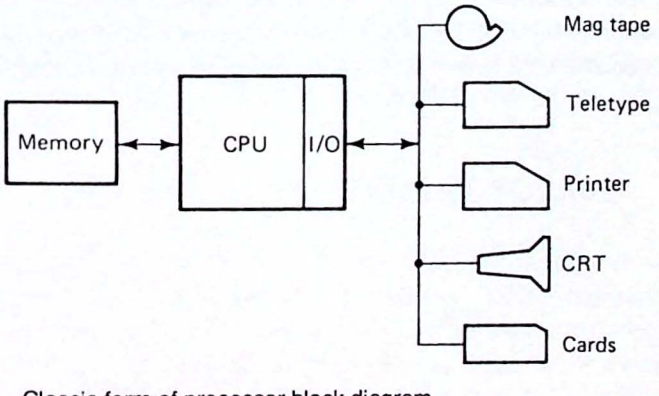


FIGURE 3-3. Classic form of processor block diagram.

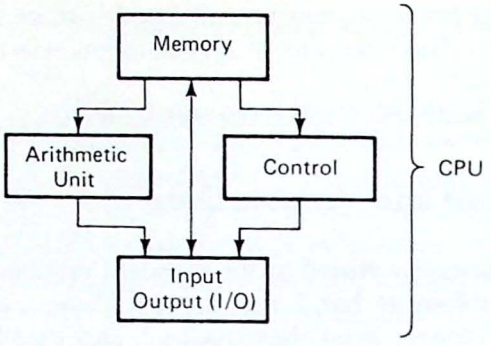


FIGURE 3-4. Functional block diagram of a processing system.

constants, variable data or characters. The major components of a binary computer are briefly mentioned. For many purposes the arithmetic unit, control, and input /output are lumped together and called a central processor unit (CPU). Other combinations are defined in practice.

3-6 PROBLEMS

1. Why are most data transfers in modern processors in parallel form rather than in series form as with some early machines?
2. What are the component parts to a 16-bit instruction of the type shown in Fig. 3-1 for a H316 computer?
3. What is the difference in codes used in the octal machines and hexadecimal machines? List the component symbols for both codes up to + 20.
4. What is the general function of each block in Fig. 3-3?
5. What functions are located in the CPU? What are peripherals?
6. If the maximum bit rate is 1 μsec, how long would it take to transfer: (a) a 24-bit word serially? (b) a 16-bit word in parallel? (Assume that only the bit rate influences the transfer.)

4

Hardware Logic Block Elements

4-1 INTRODUCTION

This chapter introduces the logic hardware. It is a general discussion, rather than detailed, to the designer level. The concept of *gating* is presented, followed by a discussion of the temporary bit storage element, the *flip-flop*. Gates and flip-flops combined form higher-level logic components, such as counters, shift registers, adders, decoders, and the like. Sections 4-6 and 4-7 describe several of these higher-level components, inasmuch as computers are usually described as a collection of these functional blocks. The differences between gate forms is most easily explained by relating the gate output to the signal occurring at the time on the input lines. Generally, a truth table is a tabulation relating the *true* or *false* condition on the output to all possible logic level combinations that may occur on the input lines.

The chapter continues with a short discourse on other digital circuits necessary for a practical system. Included in this discussion are one-shot multivibrators and level shifters.

Logic circuits belong to families where all family elements will operate when connected to other elements of the family. There are other families, and only in rare instances will elements of different families work together. Usually some special interfacing circuits are required to adjust signal characteristics between logic elements of different families.

4-2 LOGIC HARDWARE FORMS

The logic gates in early machines of the 1950 vintage consisted generally of vacuum tubes, resistors, capacitors, pulse transforms, semiconductor diodes, etc. In the later 1950's the vacuum tubes were replaced by transistors, and power levels for operation where reduced drastically. First of all, transistors operate best at lower voltage supply levels and require no filament power. In the early 1960's, the first inte-

grated circuits appeared on the market as the Fairchild DCTL logic. Although this was a great advance over transistor logic, it still presented design problems. Following rapidly then were the RTL families, the TTL families, and ECL, all with somewhat better performance capabilities. Up to 1974, the TTL logic in its various forms (ICs and MICs) has been the most popular logic form. It has been manufactured and used widely. With TTL (and also MOS logic), it has been possible to pack more gates within the same package and thus integrated circuits (ICs)) where born. A single MOS gate occupies less space on a chip (the internal circuit-carrying component) than a TTL gate; thus, more component circuits can be put in the same chip area, about 0.1 inch square. Typically, P-MOS logic is not as fast as the bipolar logic forms, such as TTL or as the higher speed ECL.

A newer form of MOS logic has recently become popular because of its extremely low power consumption when it is not being switched. This is the Complementary MOS (CMOS) introduced originally by RCA. It is popular where a minimum of power consumption is necessary.

I^2L logic under development is functionally the same as TTL. N-MOS logic is approximately as fast as standard TTL.

Intergrated circuits (ICs) were born when it was decided to package more than a single gate within a single package. Before long, IC packages contained combinations of complete circuit (MSIC), such as four-bit counters, four-bit decoders, a complete four-bit adder element, an eight-bit multiplexer, etc. Reference should be made, of course, to the catalogs of such well-known semiconductor manufacturers as Texas Instruments, Fairchild Semiconductors, National Semiconductor, Motorola, Signetics, and others.

Large-scale integrated (LSIC) circuits are now manufactured for special cases, but until 1972, the yield from these circuits was too low to make them a practical form of logic for general design. An example of an LSIC is a microcomputer.

4-3 LOGIC GATES

In a digital system, gates emit or block the flow of digital logic level signals. There are two basic types of gating provided. Coincident gates (sometimes termed the AND function gage) control the data flow to beyond the *gate*. In this gate form, *all input levels* must be enabled with a *true* signal for an output to be transmitted through the gate. The second type of gate is the OR function gate, which transmits if a signal or if *any* of its inputs is *true*. In early computers, AND gates and OR gates were each distincitve in form. Today, a single gate form can perform the AND, OR, and inverter function. In addition to AND function and OR

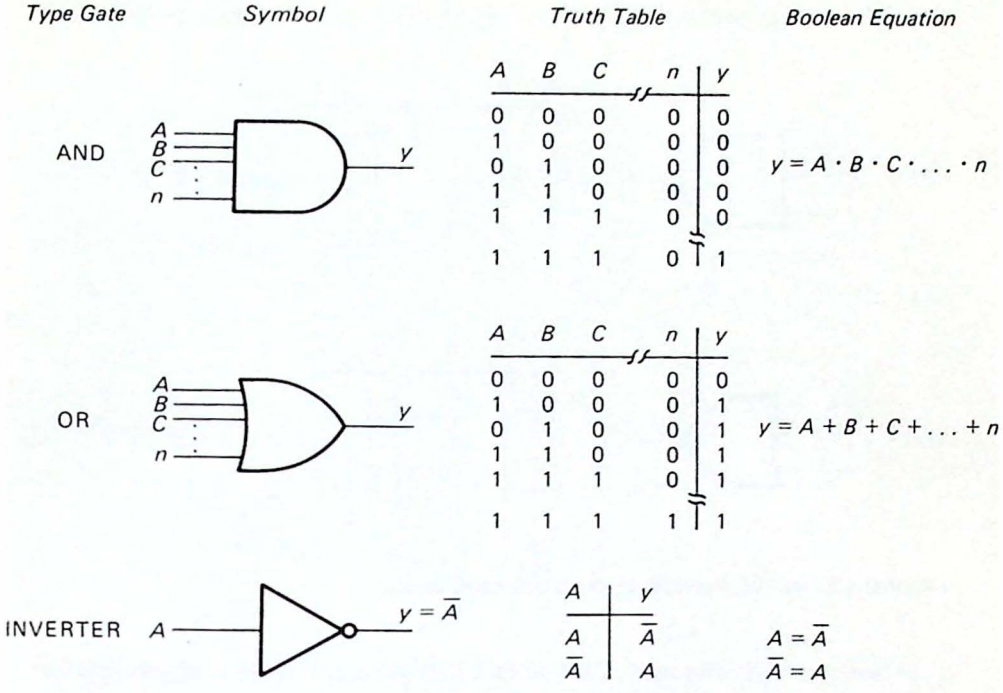


FIGURE 4-1. Logic symbols and truth tables.

function gates, the inverter element is required to implement digital logic. This element complements a logic signal. A *true* level (1) is converted to a false level (0) and a false level (0) is converted to a true level (1). The inverter, together with the AND and OR function gates is sufficient to build *all* gating functions. Symbols and truth tables for the elements just described appear in Fig. 4-1. A *truth table* lists all combinations of input signals. For those combinations providing a *true* (1) output, a (1) is listed in the output column. A Boolean equation expresses the same information. Figures 4-1, 4-2, and 4-3 list only elemental cases.

It is common practice in modern computers to employ one type of basic element for all three logic functions—AND, OR, and INVERSION. The NAND gate (or the NOR gate) is the basic element. These gates are available with a wide variety of input configurations (1, 2, 3, 4, or 8 inputs). The truth table for the NAND gate appears in Fig. 4-2. Note, for the AND function, all inputs going *high (true)* cause the output to go *low (true)*. A following NAND gate connected at the low output (where *true* is *low*) performs the OR function. This is why NAND logic can perform both AND and OR functions by itself.

A single input NAND (or single input NOR) is the inverter element. Another gating element occurs so frequently that it is deferred as

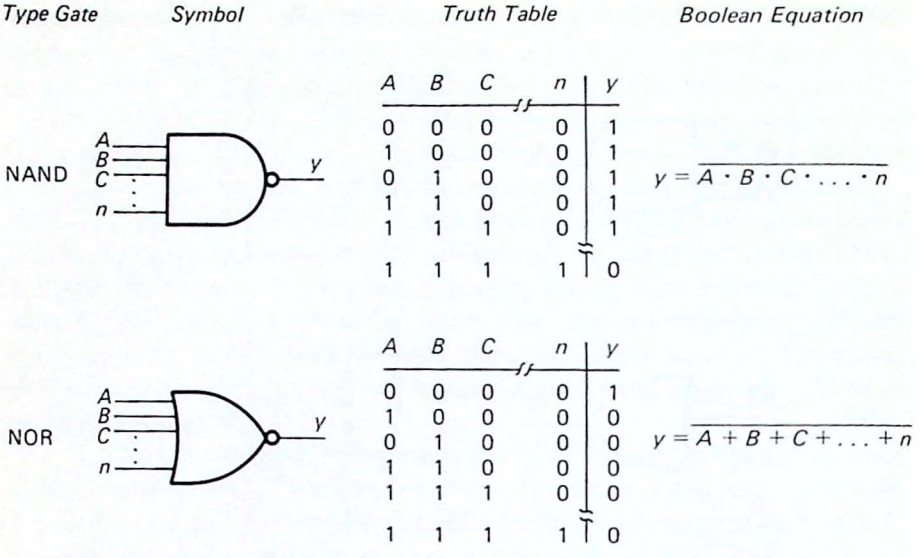


FIGURE 4-2. NAND and NOR truth tables and symbols.

a distinctive logic element. This is the *Exclusive OR* gate, which has the truth table and symbols of Fig. 4-3. The *Exclusive OR* gate is employed primarily to sense when two logic signals are alike or different.

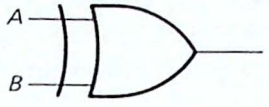
4-4 FLIP-FLOP

The flip-flop is a logic circuit that assumes one of two positions defined as SET and RESET. It has two inputs (SET and RESET) normally that allow the circuit element to be triggered into the SET or RESET state. The dual outputs are complimentary logic levels (TRUE and FALSE). In the SET state, the Q output will be TRUE while the $\bar{Q}$ output will be FALSE. In the RESET state, the Q output will be FALSE and the $\bar{Q}$ output will be TRUE. Figure 4-4 diagrams a typical flip-flop. A popular version (the J-K flip-flop) has two modes of triggering:

Mode 1. The circuit is SET or RESET by pulsing the S or R terminals.

Mode 2. The circuit is SET or RESET by placing a TRUE level on J or K and applying a clock pulse to C.

The latter mode allows all flip-flops so connected to the SET or RESET at the same clock instant.



A	B	y
0	0	0
1	0	1
0	1	1
1	1	0

$y = A\bar{B} + \bar{A}B$

FIGURE 4-3. Exclusive OR element.

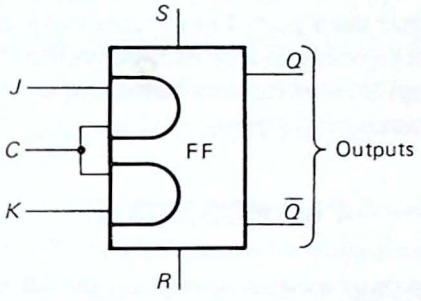


FIGURE 4-4. Diagram for a typical flip-flop (J-K).

4-4. 1 Flip-flop Controlled AND Function Gates

Probably the most common application of a flip-flop is to control the transmission of an AND function gate. In Fig. 4-5, if the flip-flop is SET the gate will transmit the logic level on line A. If the flip-flop is RESET, signal A is blocked regardless of its logic level.

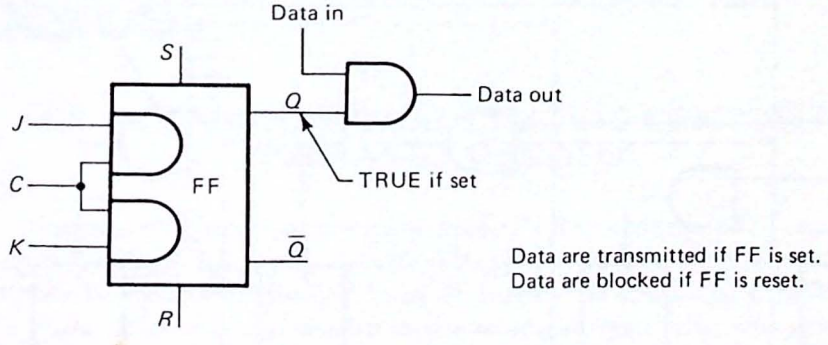


FIGURE 4-5. Data transmission controlled by a flip-flop.

4-5 COUNTERS

Flip-flops connected in tandem so that the following one in line will change position (SET to RESET or RESET to SET) when the proceeding one goes from (0) to (1) polarity form a pulse counting circuit. Fig. 4-6 illustrates the basic concept.

The circuit effectively counts pulses fed to the input. The state of individual flip-flops in the chain records the pulse count in binary number form. Adding another flip-flop increases the circuit count capability by an additional power of 2. By providing feedback paths between elements, counts other than pure binary counting are forced. Counters can thus be designed to count to any number rather than only powers of 2. Fig. 4-6 employs an increasing number of input AND gates to generate *toggle* inputs to successive stages.

4-6 REGISTERS

Flip-flops, as storage elements, create other important digital components, registers being a most important class. Generally, a register stores temporarily a binary number. This number is held until it is no longer of value. Then the register is cleared to all zeros, etc. Specifically, there are many possible types of registers as follows:

Type 1 (Buffer). This register is loaded in parallel (one flip-flop per bit) and will transmit its contents out the same way—parallel. Its major use in digital equipment has always been to provide data storage to activate displays or, more generally, to provide a temporary storage of a binary word.

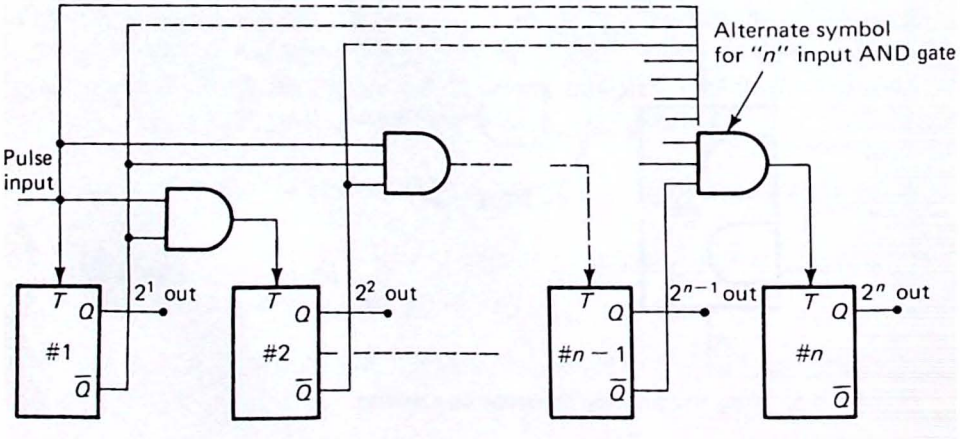


FIGURE 4-6. Operation of a multistage bit counter. A "toggle" flip-flop can be fashioned from a J-K flip-flop. It "toggles" (flips over) whenever a pulse reaches the "T" input.

Type 2 (Shift Register). This class is a very general one. Shift registers may be loaded by data in serial form and unloaded in serial form, but at the same data rate, or at another depending on the frequency of the *shift* clock applied to the shift register. Also, depending on design, it can be loaded (or unloaded) in parallel. Thus, this register is a very flexible and versatile tool. The difference between a counter and register is that a counter creates a binary number by counting input pulses, whereas a register temporarily *stores* an existing binary number, and may perform modifying operations in it, such as shifting it to the right or left. Registers properly controlled can:

1. Multiply binary numbers by 2 for each bit left shift.
2. Divide binary numbers by 2 for each bit right shift.
3. Translate parallel binary numbers into serial binary numbers and vice versa.
4. Transfer binary numbers between asynchronous systems.
5. Provide many special purpose functional operations, such as:
   a. A binary number *accumulator* register used for binary additon.
   b. Binary to BCD (or BCD to binary) number conversion hardware.
   c. Error protection and detection logic for ensuring accurate binary serial data transmissions.
   d. A special form of sequential control circuit—a "Ring" counter.
   e. A method to generate pseudo-random number codes.

Figure 4-7 summarizes the major types of shift registers. The variety of types, of course, is a function of the logic gating associated with the flip-flop stages.

4-7 COMBINATIONAL CIRCUITS COMPOSED PRIMARILY OF GATES

Groups of gates can perform prescribed functions. For example, the logic of Fig. 4-8 decodes specified bit combinations. Here four binary bits may be decoded in any one of 16 ways. The gates decoding 0000, 0010, 1101, and 1111 are shown in the example. Naturally, this general cirucit is named a *decoder*. Much of existing gating logic in digital systems is involved in this function in some way or other.

Another typical combinational gating circuit is shown in Fig. 4-9. This is termed a *full-adder*. The circuit produces the binary sum and carry

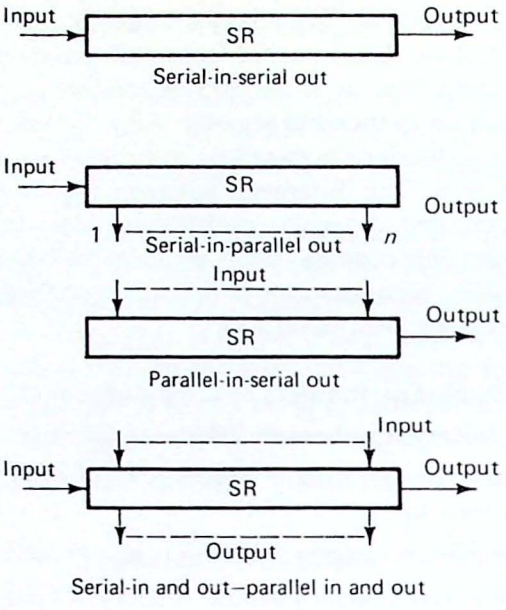


FIGURE 4-7. Types of shift registers.

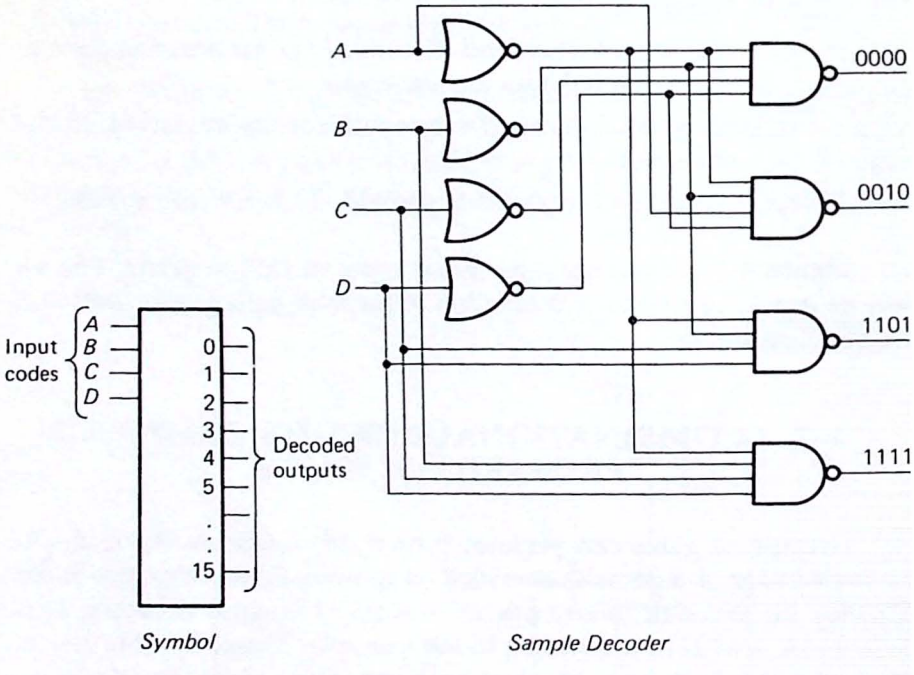


FIGURE 4-8. Decoder circuits and symbol.

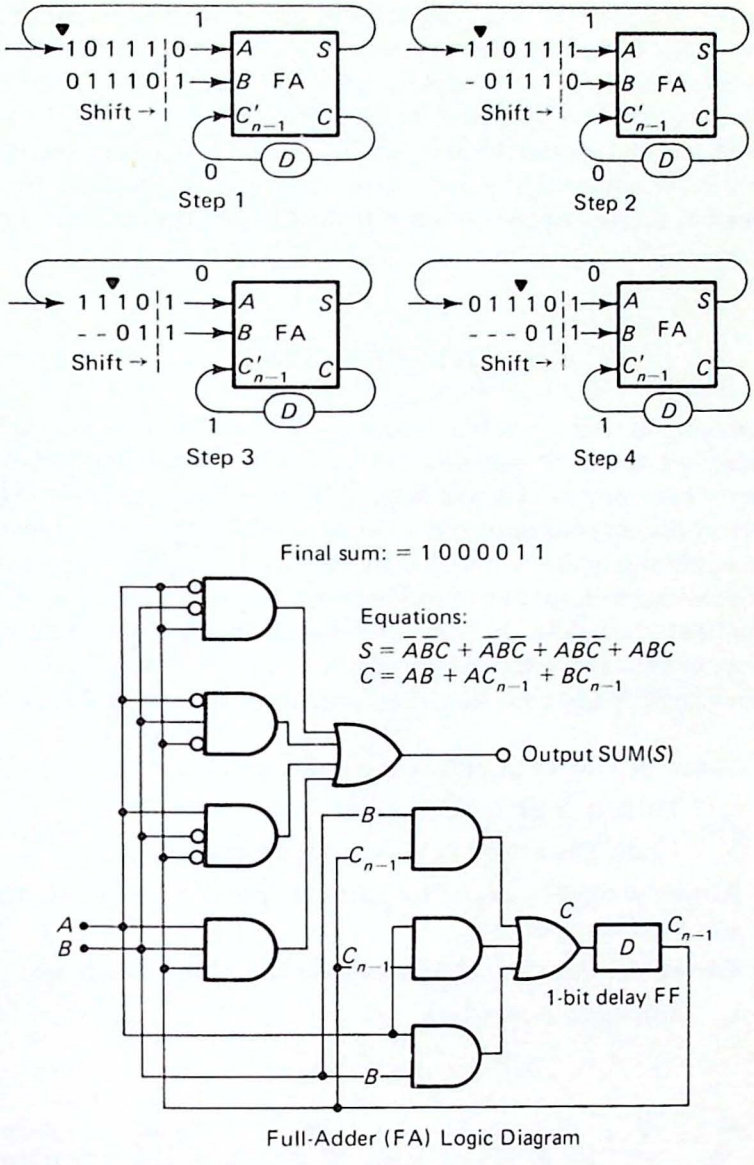


FIGURE 4-9. Function of a full-adder.

bit for the process A (1 or 0)+ B (1 or 0)+ carry (1 or 0) from a previous addition step or bit position. The full-adder was originally employed in serial addition (a serial bit stream) and in this case a delay of one bit time using a flip-flop was employed to add the carry to the next bit sum. Numbers A and B are added bit by bit from LSB to MSB along with any *carries* from the previous bit addition.

Groups of full-adders in parallel will add two binary numbers in parallel, as many bits at one time as there are individual full-adders. The total addition process is not complete until the carry pulses from lower order bits have had the opportunity ot *ripple* all the way to the left. A parallel-adder is shown functionally in Fig. 4-10. Here every bit of numbers A and B are added at the same time, along with the *carry* from the next lower bit. *Carries* ripple upward from LSB in the direction of the MSB.

4-8 INTERFACING

Interfacing at times involves connecting together two normally incompatible electrical-logic systems. Because logic signal levels between two equipments can be designed to be different, because logic (1s) and (0s) can be coded in different forms (logic levels or pulses), because at times the equipments are connected by cable that requires special line-driver circuits and terminations, and because the codes employed in the two equipments will differ for one or several of these reasons, special interfacing circuitry will often be required.

Interfacing circuits generally then will accomplish the following:

1. Match two logics of different characteristics:
   a. Pulse to logic level or vice versa.
   b. Code conversion between device codes.
2. Format changes—serial to parallel, parallel to serial, parallel word to character, etc.
3. Provide for electrical characteristics matching, such as:
   a. Impedance matching.

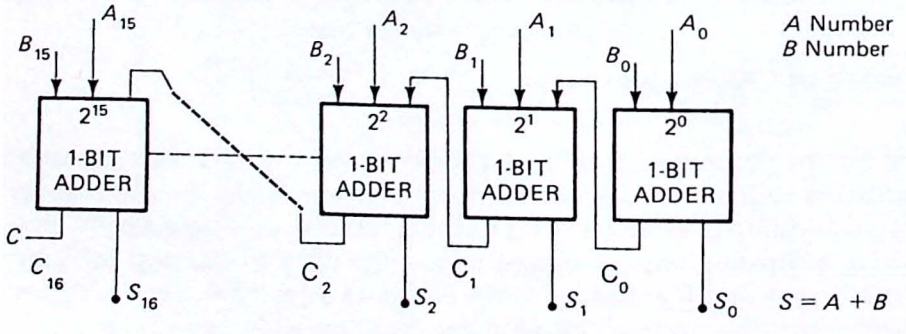


FIGURE 4-10. Parallel adder block diagram.

b. Line drivers, line receivers.
c. Isolate ground.
d. Reduce noise and interference pickups.

4. Provide for both control (addresses, strobes, responses) and data paths.
5. Provide for discrepancies in timing between the two interfacing devices. This usually involves a buffer data register in some form.

4-9 CONVERSION OF LOGIC TYPES

Internally within machines level logic is a most popular form to represent TRUE (1) and FALSE (0) levels. Essentially, the integrity of this logic form is good within the processor main frame. However, because ground levels can shift between equipment spaced at a distance, and because transmission problems become more severe, level logic is often converted to other forms before transmission over several feet (say greater than 20 feet) of cable. Manchester and pulse logic (see Fig. 4-11) are two forms that permit capacitor or transformer DC isolation, and thus ground shifts between equipments will not be detrimental to logic operations. Another problem that occurs in logic transmission is that of noise pickup (cross talk) between data lines. This problem is usually solved by transmitting logic signals in differential form; that is, a positive

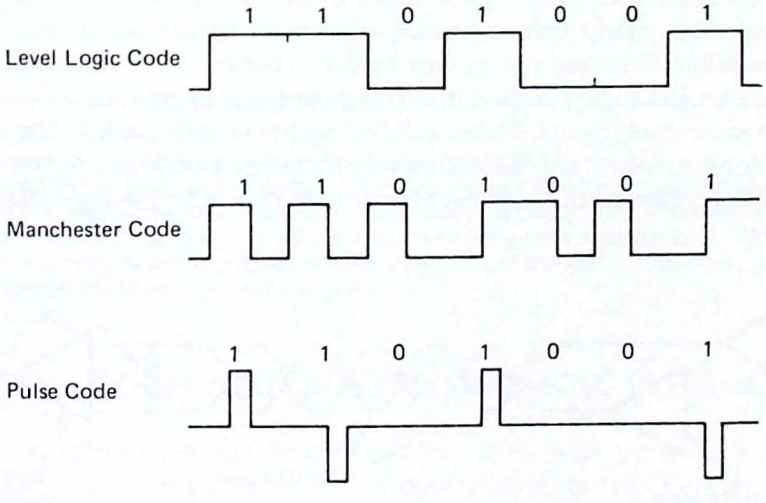


FIGURE 4-11. Corresponding forms of level, pulse, and Manchester data codes.

signal is transmitted down one wire and an opposite negative signal down another. The two wires than are twisted together, which in effect cancels any radiation of the logic signals to other wires in the same cable. At the receiver end a resistor jumpers between the two wires. This resistor absorbs the signal energy, preventing undesirable reflections from becoming established on the lines. This system of data transmission works well for logic transmission up to a few thousand feet. Ground shifts between equipments of a couple of volts will not impair logic operations, if transmission is via twisted-pair cables. For possible ground shifts of higher levels, only complete ground isolation is usually reliable enough. Figure 4-12 illustrates this form of data transmission.

Another logic circuit not mentioned previously, but of great importance, is the one-shot multivibrator. This circuit, when energized, stays *true* for a specified period of time. It is employed to generate timing delays or to shorten pulses to a standard length. One-shot multivibrators are most useful for generating timing strobes, or for adjusting for delays in interface timing.

Equipments of different manufacture will often have logic levels defined differently. Thus, circuits are required to correct for this incompatibility. Examples of logic level shifters are given in Fig. 4-13.

Other incompatibilities exist in timing between different equipments. Computers have microsecond timing cycles. Electromechanical peripherals (printers, tape readers, punches, etc.) operate in a millisecond environment. Thus, timing conversions are occasionally required between a computer and its peripherals. Data words are entered into data buffers, and the contents are held, while a print (a punch) strobe, for example, is given the peripheral requesting that it accept the data word for processing. Once started in its operating cycle, the peripheral takes many milliseconds to function. In the meantime, it is not permissible to reload the device buffer register. As a consequence, the computer has to be notified that the peripheral is temporarily indisposed to more data inputs. Thus, a BUSY signal is sent back to the computer to lock out any more data transfers from the computer to the PCU data buffer register until the present word (or character) is processed.

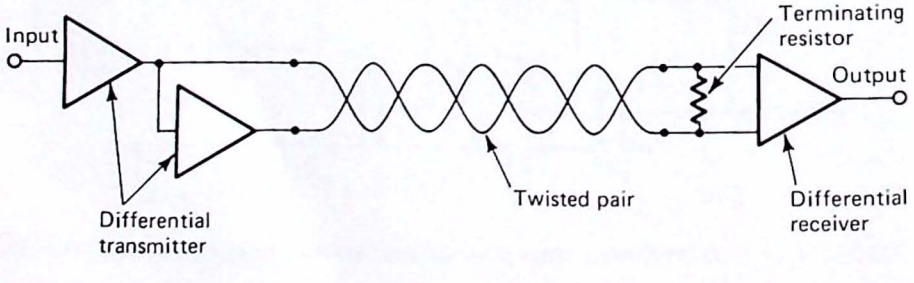
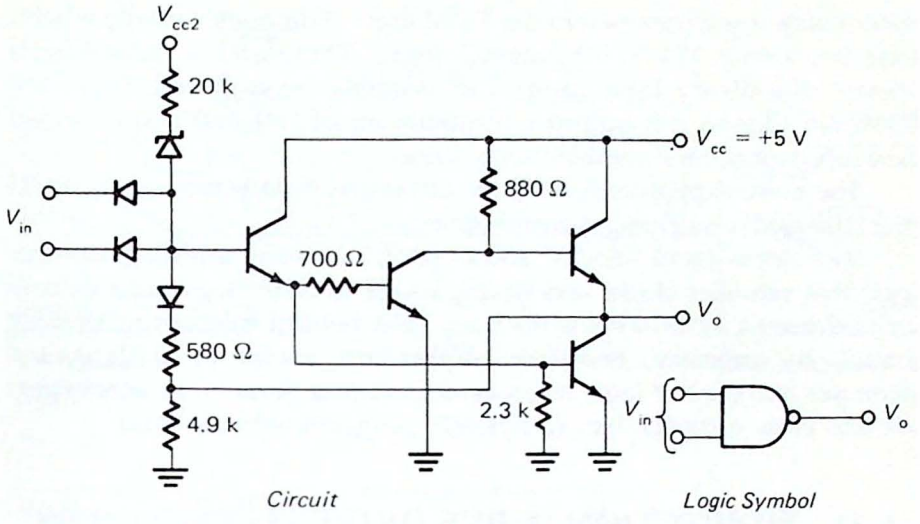
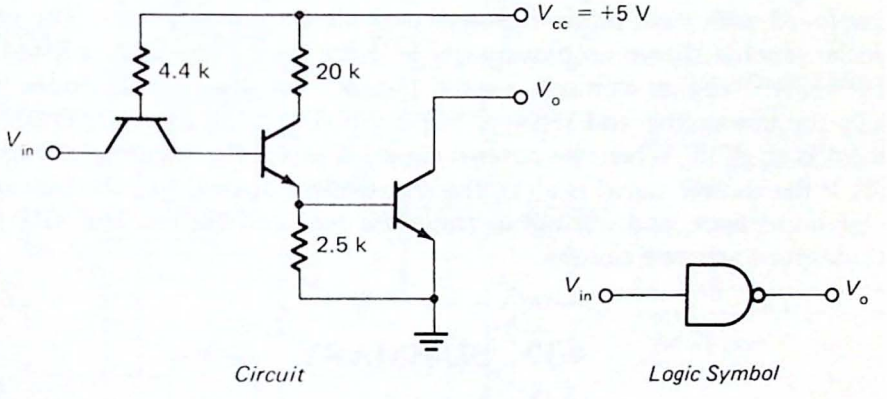


FIGURE 4-12. Differential data transmission.



(a) High to low (MOS to TTL) 8T18–2 circuits per package.



(b) Low to high (TTL to MOS) 8T90–6 circuits per package.

FIGURE 4-13. Logic level shifters. The circuit of FIGURE 4-13a (high to low level shifter) employs a higher level supply voltage than the standard TTL +5 volts on the input gate. This causes V to vary between 0 volts and +5 volts, since the output circuit is supplied with 5 volts. The reverse is true of the 4-13b circuit. Here the input gate is supplied with a TTL level voltage of +5 volts, while the output driver voltage is that of the supply to be driven, and V_o can vary from 0 volts to as high as the V_{cc} level.

4-10 ROMs AND p/ROMs AS LOGIC ELEMENTS

Typically over the years logic has been built up from individual logic gates or groups of gates and other elements. The development of ROMs and p /ROMs (p /ROMs being alterable by the customer, whereas ROMs are not) has changed the picture for the future to a large extent. A combination of binary bits addressing a ROM will select any desired bit

pattern output programmed in the ROM logic. This is essentially what is done by normal TTL combinational logic. Therefore, a considerable amount of ordinary logic gating can possibly be replaced by an LSI ROM. An address will output a combination of (1s) and (0s) as would the equivalent normal combinational logic.

The more expensive p /ROM is attractive if patterns for the ROM are not settled, and changes could occur.

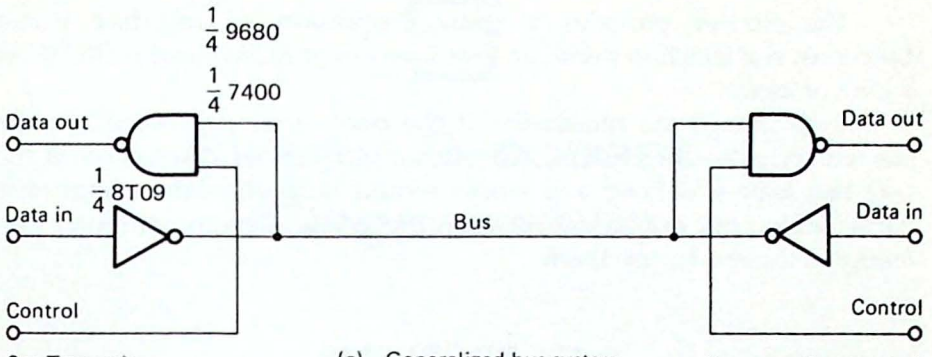
By a clever use of "clocks" and a "latch" module, timing generators (logic that provides clocks and timing levels to other logic) can be constructed from a ROM. This is the basic idea behind microprogramming control. In summary, therefore, ROMs are useful as both control memories and control logic in processors (being somewhat less expensive and more compact than completely programmable RAMs).

4-11 BIDIRECTIONAL BUS DRIVERS /RECEIVERS

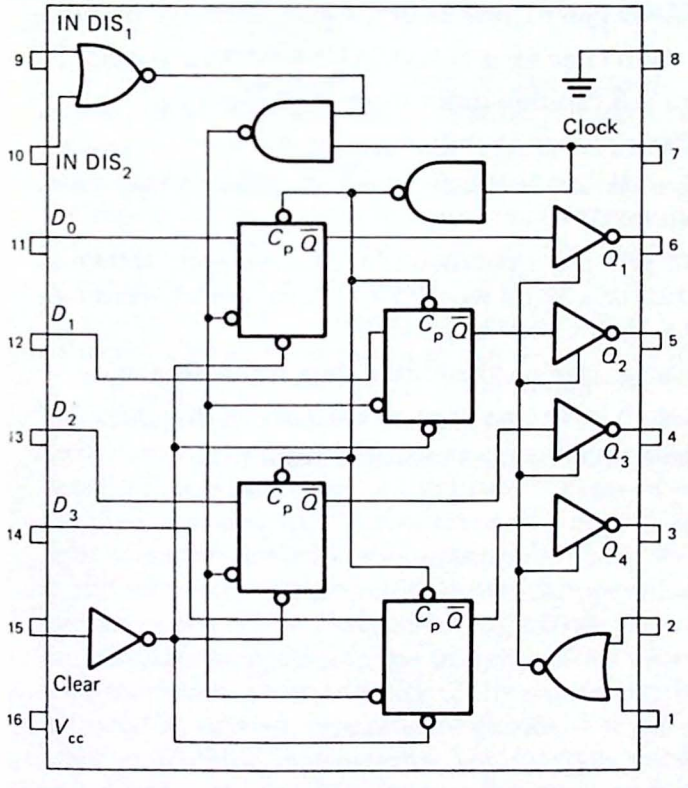
Figure 4-14 is a diagram of a bidirectional bus receiver /driver now employed with most minicomputers and all microcomputers. The particular scheme shown employs parts of three logic elements as listed in the figure. Another element, a quad D-type bus flip-flop, includes not only the transmitter and receiver but a flip-flop latch as well. This element is an 8T10. When the control signal is a (0), the receiver is biased off. If the control signal is a (1), the transmitter appears to the bus as a high impedance, and will not degrade the received signal. The 8T10 includes four separate circuits.

4-12 SUMMARY

This chapter introduces some of the aspects of digital logic at the system level. It does not discuss logic design per se. The concept of coincident gates (AND function) is explained along with gates performing the OR function. The fact that one form of gate (NAND or NOR) can do both AND or OR functions exists, and this is explained also. As a temporary storage element, the flip-flop excels. It, along with gates, can perform higher-level logic functions. A single flip-flop coupled to an AND function gate serves as a rapid-response relay for signals passing through the gate. Flip-flop elements comprise register circuits, counters, and other necessary building blocks of a digital system. The basic flip-flop circuit appears in at least two basic forms (R-S and J-K), which can be modified by external logic and connections to make other forms (toggle, D, etc.). The chapter continues with a short discussion of two all-gate combination logic circuits, the decoder and full-adder.



(a) Generalized bus system



Input Output Table

D_n	IN_{DIS}	*OUT_{DIS}	**$Q_n + 1$
0	0	0	0
1	0	0	1
X	1	0	On
X	X	1	High z

*Before clock
**After clock

Out disable 1

Out disable 2

(b) 8T10 logic pac with flip-flop storage included

FIGURE 4-14. Bidirectional bus drivers/receivers.

The chapter provides a short discussion of interface circuit functions, and concludes with an examination of ROMs and p /ROMs as a form of logic.

Even though the remainder of the book does not delve into the elementary gates themselves, the author felt a brief discussion of the way that logic is defined and works would help alleviate a linguistics barrier all too real in practice between those who design computer systems and those who use them.

4-13 PROBLEMS

1. Why is there usually a saving of logic elements if all are members of a compatible family?
2. What is a NAND gate? Draw an AND gate with five inputs.
3. What is the truth table for a NAND gate with four inputs?
4. How does an S-R flip-flop differ from a J-K flip-flop?
5. Describe different types of shift-registers.
6. What changes should be made to the counter of Fig. 4-6 to make it count to 32?
7. How do you package a parallel-adder of the type shown in Fig. 4-10 to handle a 32-bit word? How many packs will it take if each pack adds four bits in parallel?
8. What is the advantage to differential data transmission?
9. What is the truth table output of an exclusive-OR gate?
10. Where are level shifters most probably used?

5

Binary Processor Organization

5-1 INTRODUCTION TO BINARY PROCESSOR ORGANIZATION (First Generation Minicomputer)

In essence, a digital computer performs the same basic operations as a common desk calculator. It adds, subtracts, multiplies, and divides; however, it performs these operations at a speed of perhaps a million times greater than the desk calculator and its human operator. It is also possible to command the computer to perform a sequence of calculations without human intervention. This planned sequence is called a *program*. A program must be planned, carefully written, and debugged of all programming errors before the machine will execute the calculations correctly. Chapter 3 introduces computer organization.

To perform a sequence of mathematical computations under program control, the large-scale computer must possess some features that are not normally a part of a desk calculator. One of these is *memory*. the data (numbers) used in the calculations must be stored temporarily until the program says they are to be used—now! Also, the program, in the form of *machine instructions* (sometimes called *commands*), must be stored in the memory, if automatic sequential operation is to be obtained. A *machine instruction* is a code that commands the computer to calculate or transfer data according to the program. The memory organization differs with machines. *Data* and *instruction* words may be stored in a common memory or in two separate memories, one allocated to data and the other to machine instructions. The machine itself is so arranged that it will examine the machine instructions one at a time in order and execute the specified calculation or transfer of data. The machine instructions generally call out where in the *memory section* the data are located, what calculation is desired, and where to put the results of a calculation.

A block diagram of basic computer organization is illustrated in Fig. 3-3. The *arithmetic unit* performs *all calculations*. It is subject to con-

trol by the *machine instructions*. Data are moved to and from the *memory* by command of the *machine instructions*. The *memory* is, therefore, the storage place for *data* and also for *machine instructions*. Each cell where a *data* number is stored in the *memory* is identified and given a location number, called an *address*. This permits the *machine instruction* to call for the *data* by asking for the contents of cell number XXX, etc. It also permits the *data* to be stored at a particular cell, identified by an individual address number, XXX . . . . Cells are usually numbered in sequence beginning at 000 . . . proceeding up to the maximum cells provided. *Data* and *machine instructions*, when stored in the memory together, are either coded so that the machine recognizes one from the other, or they may be stored separately with the machine having a prior knowledge of where the *machine instructions* and *data* are stored.

The unit marked *control* (Fig. 3-3) is the portion of the computer that accepts the machine instructions in sequence, decodes them, causes the transfer of data into the *arithmetic unit*, commands the arithmetic operation, and transfers the computed data out to the desired memory cell. It is in reality the *commander* of the computer, since it also controls the *input/output* unit.

The *input/output* unit provides access to the outside world. It allows the connection of tape recorders, paper tape punches, paper tape readers, digital typewriters, printers, keyboards, or other devices to the computer. Figure 3-4 outlines the I /O's function. Typical devices are listed in Table 5-1 and are identified as input, output, or bidirectional (input and output).

Although the machine instructions emphasized so far transfer data and commands between memory and the arithmetic or control section, there will be many other instruction types in any computer repertoire. These include instructions that transfer data from register to register in the machine, instructions that shift the word to the left or right in a given register, instructions that modify other instructions that control input /output including the peripheral equipment, instructions that mask data, instructions that perform tests—just to name a few. The decision on what to include in an instruction repertoire is one that requires close cooperation and study between the programmer, the designer, and even the eventual user of the machine.

5-2 MEMORY

5-2.1 Core

In most of the first generation minicomputers, the memory was magnetic core. Figure 5-1 is a block diagram of this memory form. In future generations the shift to MOS types has been nearly universal. Core

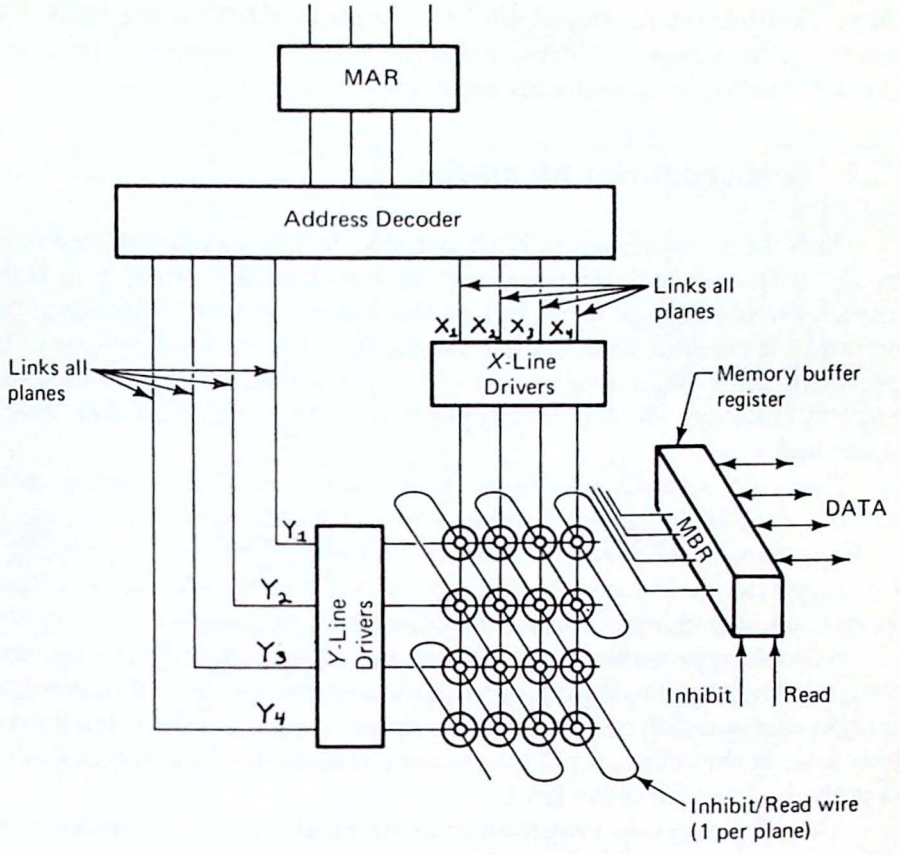


FIGURE 5-1. Block diagram of a core memory.

memory provides separate driver lines through each corresponding core in each plane and works on the principle that cores located at the junction of a unique X-Y driver is either stored into (1) or (0). The word at a particular X-Y location is stored or read in terms of (1s) or (0s), the storage and reading process taking place in parallel. Current in X_2 driver line and Y_3 driver line addressed via the memory address register (MAR) and address decoder will cause word X_2-Y_3 to be stored or read out. A (1) is stored by exciting S_2 and Y_2. An (0) is stored by exciting X_2, Y_3 and an inhibit line /read line that is unique to each plane. The inhibit signal cancels the effect of either X_2 or Y_3. The bits are usually arranged in planes. For example, a 1024-bit word machine would have 32X driver wires and 32Y driver wires. The individual planes are stacked in series, and each has its own inhibit /read wire [(0) write process], and this line (I) is excited only to store (0s [X and Y store a (1) X + Y − I stores a (0)]. The same wire is often used in the read process, where it is connected to the memory buffer register (MBR). For the read process the currents X and Y are reversed, which causes a readout of (1s) only. Thus a 16-bit parallel

word is emitted from (or stored into) the memory stack via the MBR. The memory address register (MAR) provides the X-Y address. This is also generally true for semiconductor memories.

5-2.2 Semiconductor Memories

With the development of high-density, large-scale integrated (LSI) circuits, it is now both economically and technically feasible to build memories in circuit logic form. Before the advent of this technology, the core and plated-wire forms ruled practically unchallenged—except in specialized cases. Beginning in 1971–72, semiconductor memories began to openly challenge the magnetic types that had been standard for over a decade and a half.

There are basically two forms that a random access LSI memory can take. One is the *static cell* type, where bits are stored in a form of flip-flop. If these cells are of the MOS-FET variety, there is probably a need to refresh their contents, since some MOS-FET cells store (1s) and (0s) as capacitive charges and tend to discharge with time.

A second type is the dynamic form, where the stored data are circulated back in (output to input), as via a shift register. This, of course, can be organized as a shift of a few bits in sequence and, in effect, is a form of delay line; or the cells can periodically recirculate on themselves (sort of an individual one-bit delay line).

The advantages to semiconductor or random-access memories lie in their promise for less expensive and faster access memories. Since the elements are effectively ICs, they interface CPU logic with less complexity and usually reduce the need for unique, special power sources for the memory stack itself. IC element forms can be attractive for memory applications. Other developments in the semiconductor memory field, such as charge coupled devices (CCD), appear to offer great potential for the future. A simplified block diagram for a semiconductor memory is shown in Fig. 5-2.

Bipolar ICs, although somewhat faster than MOS memory devices, suffer from the requirement of using more space per memory element on the chip, thereby reducing total memory size per chip. It is expected that IC techniques will see further advances as LSIs become more common and, in effect, both bipolar and unipolar devices will be widely employed.

5-3 CENTRAL PROCESSING UNITS (CPU)

A CPU is the collection of logic (gates, registers, etc.) that performs (under program control) the arithmetic operations (such as add, subtract, multiply, divide, compare, etc.) on the data and provides the in-

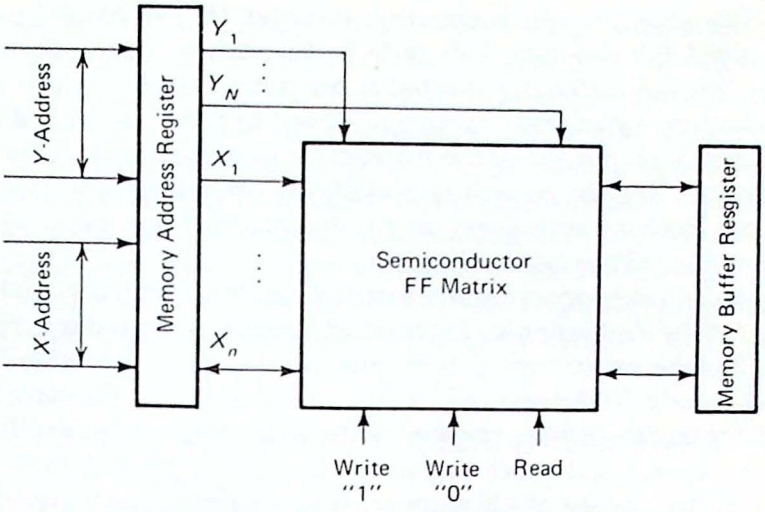


FIGURE 5-2. Block diagram semiconductor memory (RAM memory).

struction decoding and control signals for transferring data between memory, the input /output, and all other processor registers. In a simplified form, Fig. 5-3 illustrates the organization of a simple CPU. The variant bit control determines the final addressing mode—whether indexing, indirect address, etc., is to be used.

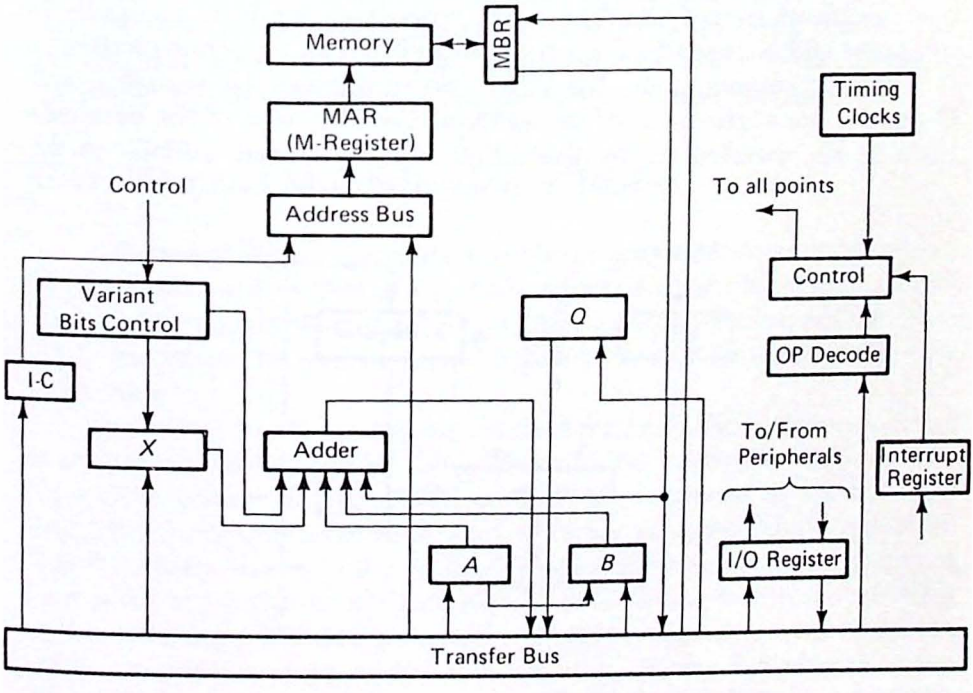


FIGURE 5-3. Random-access computer organization.

A code stored in the instruction counter (IC) is loaded into the memory's MAR (Y register). This code is the address of the selected instruction. The instruction so selected is outputted from memory via the MBR, and its operation code in turn is stored in the F register; if appropriate, the address portion of the instruction is placed back in the MAR. However, there may be an address modifying operation called for by the variant bits, such as by *indexing* or by the specification of an alternate address via the *indirect* addressing process.

Indexing of memory reference instructions is indicated by a (1) in bit position 1 of the instruction and *indirect addressing* indicated by a (1) in bit position 2 of the instruction. If both bits are (0), the instruction is to be executed directly. In the case of *indexing*, the contents of the *index register* (X) is added to the address portion of the selected instruction (both instructions address and index register contents are routed through the adder) and then to the MAR register, where the contents' resultant is now the address of the instruction to be executed.

For *indirect addressing*, the present instruction code is interpreted as the *address* of a new instruction, and the new instruction is fetched from the indicated *address*. Either of these techniques permits a modification to be made in instructions (and /or data) as currently stored. In general, the modification cycle is required first to obtain the final instruction to execute—although, in some instances, an instruction may first be executed and then indexing, etc. may be performed afterwards. *Indexing* and *indirect addressing* described here are representative forms of instruction modification methods. The exact process varies from computer to computer and is dependent upon the individual design characteristics.

For processing data, the *adder*, accumulators (A-register), and Q-register are at the heart of the system. Usually, one of the operands (data to be operated on by the CPU) will be stored initially in the

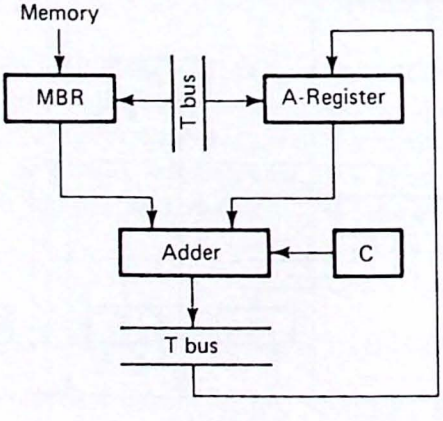


FIGURE 5-4. Parallel addition/subtraction.

A-register, and the other in the MBR. The resultant arithmetic operation will be stored on completion in the A-register (or Q-register) from whence it may be transferred by program control to where it is needed.

By adding a double set of registers, address, as a single register etc., it is possible to provide rapid double-precision addition or subtraction. Quite often, the only double-precision operations provided by hardware are double-precision transfers. The double-precision arithmetic operations are often performed via software-controlled macros (i.e., subroutines). The B-register allows a double-length word to be accessed and operated upon in double-precision operations.

5-4 ARITHMETIC UNIT OPERATION

5-4.1 Addition /Subtraction in Binary Machines

The usual practice in binary machines today is to operate at the same time on a number word in the accumulator (A-register) and the number word from a desired memory position (resting in the MBR) to the binary adder network. The resultant addition (after carries have propagated) appears in the A-register replacing the starting contents. If a subtraction were required, one of the two numbers would be replaced by its complements before adding. Most processors today employ a parallel add process, which means that all bits are added at once, followed by adding all carries generated. This is a much faster process than the original serial-add process employed in the earliest of digital machines. Fig. 5-4 illustrates the process of addition /subtraction.

5-4.2 General Multiplication /Division

All binary digital computers perform addition or subtraction via a binary adder such as that of Fig. 4-10. Multiplication and division, however, can be accomplished by direct hardware operations; or in the less costly machines, by successive additions (subtractions), and shifts via a subroutine.

In the hardware processing method for multiplication, one method is to store in the Q-register the multiplier. The multiplier bits, one at a time, or in groups of two, three, or four, are examined by special logic, and it is determined whether to add, subtract, or merely shift the multiplicand (usually stored in the MBR). The multiplication process is, thus, a controlled sequence of additions and shifts—logic in the *multiplication control* defining when the process is *complete* and exactly what multiples of the multiplicand to apply at each step. Figure 5-5 illustrates the binary-bit multiply process.

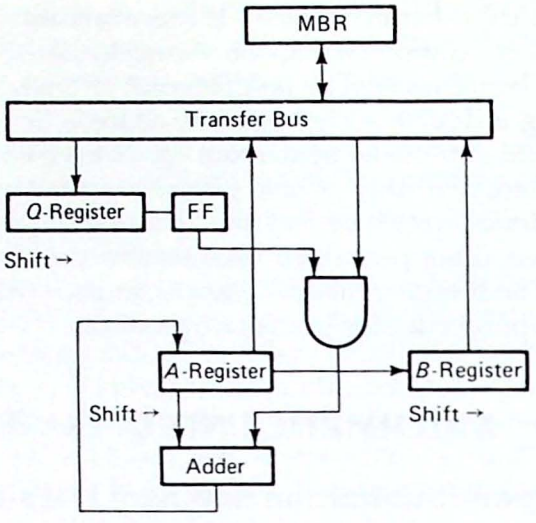


FIGURE 5-5. Parallel-bit multiply.

The binary division process is somewhat akin to multiplication, and normally requires repeated binary subtractions of the divisor into the dividend remainder. The dividend-remainder would normally appear in the accumulator. The divisor (in the MBR) is subtracted (starting at the left, as in decimal long division). If a valid subtraction occurs (dividend remains positive), a (1) is entered into the MSB of the Q-register. If the dividend becomes negative, a (0) is entered in the MSB of the Q-register. The divisor is then added back to the dividend-remainder to restore the dividend to its former value. Then the divisor is shifted to the right one place and the subtraction is attempted once more. Again, a valid subtraction results in a (1) being entered in the next lower position of the Q-register. If the subtraction is not valid, a (0) is entered, etc. The process continues the required number of steps under control of specialized control logic. After the last step, the quantity remaining in the accumulator is the *remainder* as in decimal division.

5-4.3 Double-Precision Arithmetic

The operation of double-precision arithmetic assumes double word lengths (use of two memory positions to hold a particular number). As an option in the H316 computer, a small amount of hardware and a few extra instructions are required to perform double-precision transfers of data in and out of memory and between the A and B accumulators. The arithmetic instructions are limited to double-precision ADD (or subtract); therefore, double-precision multiplication and division must be provided by programmed subroutines.

5-4.4 Floating-point Operations

5-5.4.1 General This section treats floating-point numbers. By way of definition, a floating-point number is one represented by a numerical coefficient and a power of the radix. It is a shorthand notation that enables numbers of a wide magnitude range to be represented compactly and is related closely to scientific number representation of the following forms:

$$807{,}600 = 0.8076 \times 10^{6}$$
$$80.76 = 0.8076 \times 10^{2}$$
$$0.0008076 = 0.8076 \times 10^{-3}$$

In the general form adopted for floating-point decimal numbers, the radix can be dropped; therefore, the above numbers become, respectively, in floating-point notation

$$+06, .8076$$
$$+02, .8076$$
$$-03\ .8076$$

The right-hand term is usually termed the fraction, and the left-hand portion is the exponent (or characteristic). Later a method is presented that will cause all characteristics to be positive and eliminate the awkward negative sign.

To illustrate floating-point addition, consider the two numbers

$$566 \times 10^{-1} + 6.72 \times 10^{2}$$

We know that to properly add these numbers, the decimal points must be located and the two numbers added as

$$56.6 + 672 = 728.6$$

What we have done in reality is to make the two characteristics equal by moving one or the other of the decimal points. For instance,

$$566 \times 10^{-1} \times 10^{3} \times 10^{-3} = 0.001 \times 566 \times 10^{2} = 0.566 \times 10^{2}$$

Now with the characteristics equal, the addition is

$$\begin{array}{r} 0.566\times 10^{2} \\ \underline{6.72\times 10^{2}} \\ 7.286\times 10^{2} \end{array}$$

The floating-point notation gives us the same information with regard to the decimal point location. The addition process, therefore, operates under the following rules:

1. Multiply the number with smaller characteristics by 10^n which is equivalent to moving the fraction's decimal point n places left.
2. In the actual process $n = q - s$, where q is the more positive characteristic and s is the less positive characteristic. This makes the characteristics equal and redefines the position of the decimal point of the fraction.
3. Add the fractions resulting from Step 2.

The processes above might be represented in compact form as

$$(f_1 \times 10^{s} + f_2 \times 10_{q} = 0 \ldots \underbrace{00}_{n \text{ places}} f_1 \times 10^{s+n=q} + f_2 \times 10^{q}$$

Let us now consider the problem of floating-point multiplication. Again we return to the scientific notation for guidance. Suppose the problem is to multiply two numbers of the form

$$(5\times 10^{2})\,(1.01\times 10^{-4})$$

The rules for scientific notation in this case are to multiply separately the fractions and add the characteristics. Thus, multiplying the fraction terms, we have

$$5\times 1.01 = 5.05$$

Adding the characteristic terms, we obtain

$$10^{2} + 10^{-4} = 10^{-2}$$

Therefore, the result obviously is 5.05×10^{-2}.

For division, the process is similar except that one divides the fractions and subtracts the characteristics. The multiplication process can be summarized as

1. Multiply fraction $f \times f_2$.
2. Add the characteristics a and b to obtain 10^{a+b}.
3. The final result will be $(f_1 \times f_2) \times 10^{a+b}$.

In summary, floating-point notation is merely power of 10 (octal or HD) notation implementized. However, there is one more extension adapted by most computers that contradicts our simple, well-understood example. To ensure that the characteristic is always positive within the working range of numbers, an artificial value is added to the characteristic. This could be cited as the early IBM 704 approach. For example, if the number is expected to range $f \ldots \times 10^{-a}$ to $f \ldots \times 10^{+a}$, the scale of the characteristic could be offset to artificially range from 0 to $2a$. The original notation and the corresponding offset notation are arranged below:

Original	Offset
$f \times 10^{a}$	$f \times 10^{2a}$
.	.
.	.
.	.
$f \times 10^{1}$	$f \times 10^{a+1}$
$f \times 10^{0}$	$f \times 10^{a}$
$f \times 10^{-1}$	$f \times 10^{a-1}$
.	.
.	.
.	.
$f \times 10^{2a}$	$f \times 10^{0}$

In this manner, the negative characteristics are not handled at all in the computation; but they may be restored, if ever needed, since the actual coefficient is $q = q_0 - a$, where q_0 is the observed characteristic and q is the actual value of the characteristic.

EXAMPLE:

If it is decided to handle numbers ranging from a $\times 10^{6}$ to a $\times 10^{-6}$ then 10^{6} would be added, or the characteristic would now range from 0 to 12 rather than from -6 to $+6$. For instance, $0.573 \times 10^{3} = 0.573, +3$ will become $0.573, +9$. See Chapter 11 for the Interdata Computer floating-point operation.

5-5.4.2 *Mechanization* Because of the added operations of characteristic manipulation, it generally will take somewhat more computing time to accomplish floating-point operations as compared to fixed-point operations. This is true usually whether the two types of number notations are manipulated by hardware or software.

The operations of subtraction and division were not discussed at length, since they are carried out in a manner very similar to addition or multiplication. The mechanization of the floating-point process is indicated by the flow chart of Fig. 5-6. Because the basic operations involve only addition, multiplication, compare, and shifting (in binary notation, the floating-point process can be implemented by either built-in specialized hardware or by programming. Again, the programs can be classified as subroutines. The instructions when performed in sequence will effectively perform floating-point arithmetic on numbers stored in floating-point form. Figure 5-6 defines the order of steps to perform floating-point arithmetic. It is interesting, though not necessarily unique, that the referenced figure represents either the hardware or software steps required of floating-point arithmetic.

In summary, why use floating-point? It is easier to represent 105,320,000 as +09, .105320. It is a more compact notation because fewer digits need be stored.

5-5 CONTROL OPERATIONS

5-5.1 Instruction Decoding

The decoding of a one-address instruction is demonstrated by Fig. 5-7.

As shown in Fig. 5-7, the operation code portion and the address portion of the instruction are transferred from the *memory buffer register* (MBR) and the separate portions are routed to and decoded within the two separate decoding matrixes.

One-address Instruction System (Cycle of Operations) The execution of an instruction in a one-address system logically divides into two general operations.

The first operation, called *program time*, refers to the following three steps:

1. Fetch the instruction address (usually from the instruction counter).
2. Transfer the instruction word from the MBR to the operation decoder and address decoder.

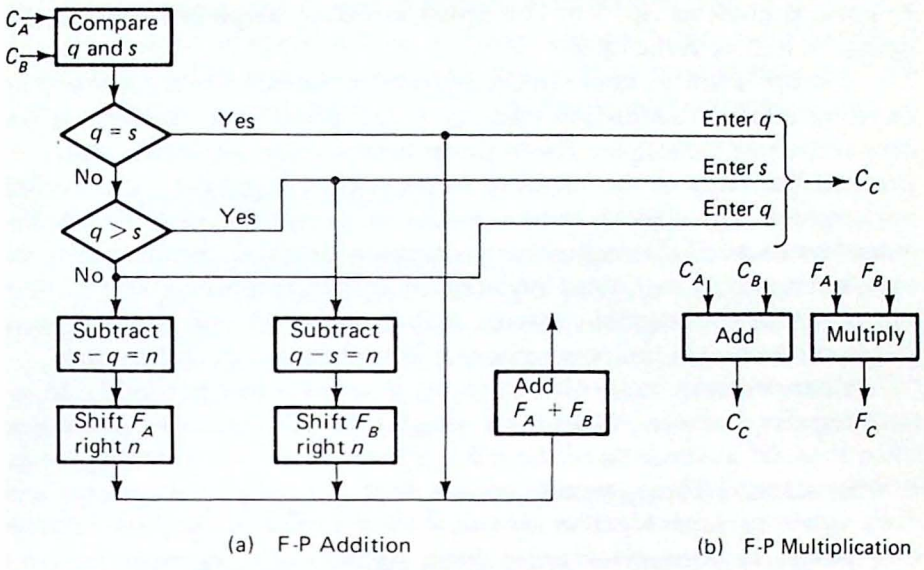


FIGURE 5-6. Floating-point operation chart.

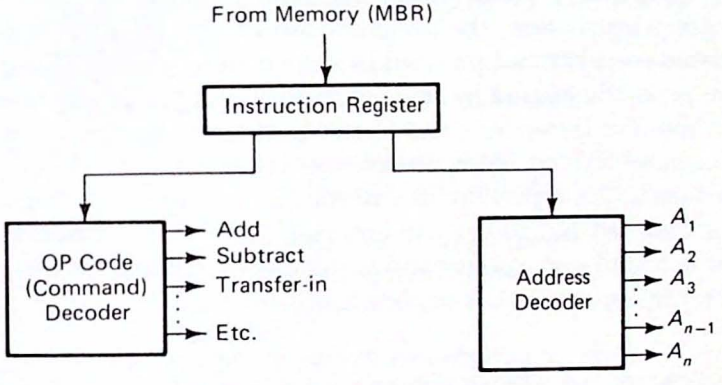


FIGURE 5-7. Decoding a one-address instruction.

3. Decode the instruction (both operation and address). This operation generally takes the same amount of elapsed time regardless of the command being executed.

The second operation, called *operation time* is that period devoted to the execution of the instruction. The length of time required for this operation is a variable. To illustrate the method and clarify the philosophy, the total operation steps, combined to perform a complete addition of two numbers, are listed in Table 5-1.

The block diagram of the process for executing a one-address in-

struction is given in Fig. 5-8. The operation flow steps are listed in the figure, as well as in the table.

The operation sequence of the *instruction counter* can be changed by inserting into the instruction counter a complete new setting via the *jump instruction* technique. These jump instructions are often stored in particular portions of the memory or in special registers, often called *index registers*. The object is to transfer the *index register* contents into the *instruction counter*. The instruction counter's original contents may be stored temporarily in a third register for future reference; that is, it is stored so that the original contents may be recalled and the computer program returned to the point where it left off in its normal routine.

Often the *index register* is employed in other ways to modify an instruction. For instance, rather than transfer the contents, the procedure often is to *add* to contents of the *index register* to the contents of the *instruction counter*. This generally means that the *instruction counter* and *index register* contents together are run through an *adder*, and the result is then put in the *instruction counter*. *Index registers* are also used in count modes as implied in the later section of programming.

By the *index register* technique, it is possible to jump from the normal instruction set to a completely different one. On the command of a second jump instruction, the program can return the *instruction counter* to its former contents and proceed in the normal manner. Jump instructions are generally caused by the occurrence of a specified result during a calculation. For instance, if the contents of the *A-register* go from positive to negative during an accumulation: If a jump is to occur, it is built into the instruction repertoire of a particular computer. If the jump condition is detected (whatever the criteria), the normal operation of the machine is halted and a command is generated calling for the change to the jump routine.

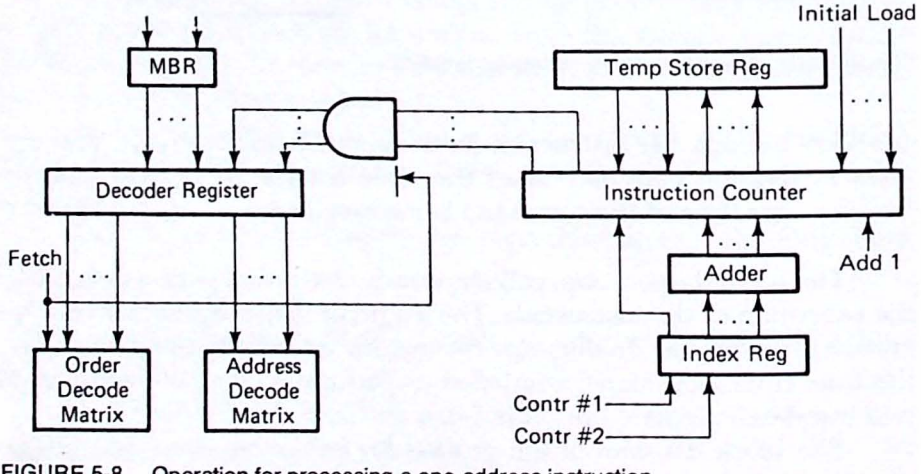


FIGURE 5-8. Operation for processing a one-address instruction.

In any event, the normal instruction is replaced with a new instruction. The simplest approach is to change the *instruction counter* contents by use of the *index register* and then proceed in a normal manner with the next *instruction counter* contents, specifying the next command.

5-5.2 Instruction Counter

This element of the control system is set initially by the programmer to the address where he desires the program to start. Initially, he may set manually by the front panel buttons the starting point program address. After each command is executed, the instruction counter is normally incremented by 1; thus, it will address in turn each succeeding instruction. For some conditional instructions, it may be incremented by two, etc.

The orderly progress of the *instruction counter* may be changed by:

1. Indexing, which means summing the contents of an *index register* to the present contents of the *program counter.*

Table 5-1. Instruction Operations Required to Perform A Complete Addition

Operation 1.	a. Fetch instruction address and transfer instruction.
	b. Move Instruction 1 from memory via MBR, and then to the operation decoder, address decoder, etc.
	c. Decode operation code and address.
	d. Transfer Word 1 to accumulator.
Operation 2	a. Fetch instruction address and transfer instruction.
	b. Move Instruction 2 from memory via MBR, and then to the operation decoder and address decoder.
	c. Decode operation code and address.
	d. Transfer Word 2 (aligned with Word 1) through the adder.
Operation 3	a. Fetch instruction address and transfer instruction order.
	b. Move Instruction 3 from memory via MBR, and then to the operation decoder and address decoder.
	c. Decode operation code and address.
	d. Transfer word in accumulator to memory address 4, etc.

Note: Steps a, b, and c are accomplished during program time. Step d is accomplished during operate time.

2. Use of an indirect address instruction, which inserts a replacement address from another memory position in the *instruction counter.*
3. Substituting for the present address in the *instruction counter* a new address retrieved from a special register.

5-5.3 Usual Control vs. Microprogrammed Control

In the design of a computer control system, it has been the practice in the past (era before 1970) to provide a unique set of control gates for each instruction. These gates were located throughout the computer wherever needed. Although this general system allows for rapid execution of instructions, it is likely to be a haphazard organization; hence, construction of control sections turns out to be a conglomeration of gates with no particular apparent order. Thus, ICs are not particularly useful for building this general logic area.

In an attempt to regularize control logic, the concept of microprogramming has been proposed and implemented. In this system, the main computer instruction (macroinstructions) operation code (op code) acts as an address to a *control memory*. The *control memory* in effect then issues a *microinstruction*, of which each bit calls for an individual operation step (microstep) in the execution of the desired machine instruction. All machine instructions are composed of specific and standard microsteps. The microstep for each machine instruction is stored in a read-only memory (ROM). The timing and routing of the microstep commands are controlled via a *control matrix network.*

By modifying the *control matrix network* it is feasible to modify the constituent microsteps in each machine instruction. New instructions can thus be created and old ones eliminated by modifying the control matrix alone—without changing other logic wiring itself. This, in effect, leads to more uniformity in control system design (see Fig. 5-9).

5-6 EXTENSION OF ADDRESS CAPABILITY

In a typical memory-addressing instruction, two bits will be employed to signify the process of *indexing* and *indirect addressing*, and four bits will be used to identify the operation (op code). The remainder of bits may be employed as a memory addressing code. For a 16-bit instruction, this leaves a maximum of ten bits for addressing—or, at most, 1024 (1K) words are directly addressable. Since even minicomputers typically have memories of 4K words or greater, it is desirable for complete programming access flexibility to provide a means for the program to address easily any word in memory. Several techniques have been employed to

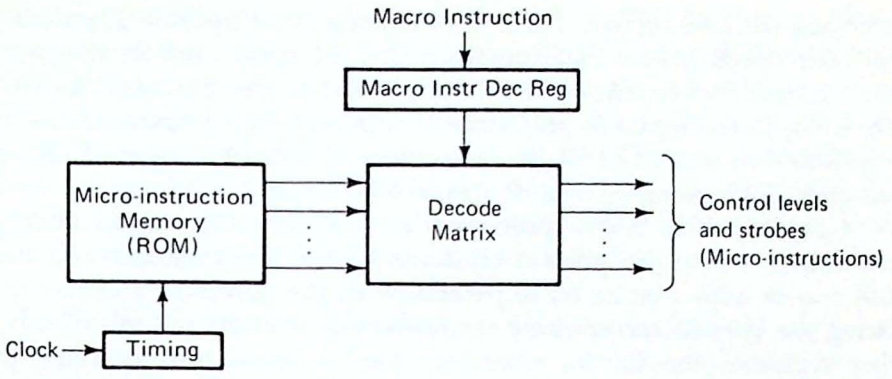


FIGURE 5-9. Microprogram control.

extend the addressing capability of 16-bit machines. These techniques include *indirect addressing,* previously outlined in Sect. 5-5.2, or one of the following methods:

1. An *indirect address* where the entire bit grouping is recognized as the address of the instruction to be executed.
2. Storing additional address bits in a register. These bits are combined with the normal address bits to give an extended address range.
3. Employing a double-length word, where the second word is recognized as the address. This, in effect, is a modification of 2 above.

In the Honeywell Inc. 16-bit control processors, one bit in the 16-bit instructions causes the processor to execute an instruction in either the 512-word sector in which the program presently finds itself, or to execute the instruction found at a corresponding address in a base sector. The instruction in the base sector can be an instruction that causes jumps into any 512-word sector of memory. Thus, by a maximum execution of two instructions, any word in a 32K memory is addressable.

The IBM 1800 uses a different scheme, where a 16-bit instruction addresses 512 words on either side of the present instruction counter setting or a 32-bit word is employed to address beyond these limits.

5-7 THE COMPLETE PROCESSOR

There are so many possible variations of a complete processor that it is impossible to generalize in any detail. In the classical description, the processor consists of an internal memory, a central processing unit, a control section, and an I/O section. Occasionally, the control panel is

tested as a separate section. There are machines that operate in pure binary code words, others that operate with octal codes, others that perform decimal arithmetic with decimal codes, and many fourth-generation minicomputers and microcomputers (μ-computers) use a hexadecimal code; that is, all possible codes of four bits are used rather than codes (0) through (9) as in the pure BCD code.

Generally, the binary processor is organized about an adder-accumulator where the present contents of the accumulator and the MBR may be added bit by bit in parallel with the resultant addition replacing the present accumulator contents. As pointed out previously, other registers (the Q, for example) are incorporated to assist in hardware multiplication and division. A simplified block diagram for the binary processor appears previously in Fig. 5-3.

Although the parallel binary processor is the most simple in logic organization, other forms have been employed. For example, exactly four-bit μ-processors organized about and operated with decimal codes. Since it requires four bits to represent a decimal code (BCD for example), the groups of four bits comprising a decimal digit are transferred in parallel together. Thus, decimal numbers of any length can be represented by groups of four bits; where the groups are processed in series. Special BCD logic or programs are designed to perform a decimal addition, multiplication, division, etc.

Octal arithmetic is performed with three-bit octal logic. In a similar manner, IBM developed a four-bit number system (hexadecimal) where the invalid four-bit code of normal BCD has been utilized as numbers.

The general organization of the octal code and hexadecimal machines imply three-bit or four-bit codes being utilized in arithmetic processes. This removes the necessity of BCD (or other) code conversions to binary for numerical processing (common to binary machine organizations). Usually six-, seven-, or eight-bit codes are employed so that alpha characters can be stored or transferred also with the same ease as coded numerics. The numeric processing logic ignores the alpha codes.

5-8 INPUT /OUTPUT

The general design of the input /output section allocates to itself some of the above functions and designates others to special controllers (sometimes termed peripheral control units—PCUs).

The input /output section usually provides for a standard path to address peripheral controllers. In the PCUs, on the other hand, addresses on their address bus can be decoded by the PCU. Usually the PCU also contains data buffer registers for accepting data from the com-

puter output data bus and /or provides data to the computer input bus for inbound data. The decoding logic of the PCU permits the individual data buffer register to be actually connected to the data buses only long enough to transfer data between a particular PCU and computer (or vice versa). Thus, a single computer input /output channel effectively may service several peripherals via the individual PCUs (see Fig. 5-10).

In addition to the obvious functions of data transfers, the PCU can decode other specific computer commands, such as:

1. Turn *on* the peripheral.
2. Turn *off* the peripheral.
3. Run the peripheral in reverse (rewind tape, for example).
4. Run a self-test program in the peripheral.

Also, the PCU may permit the computer program to test it to ascertain the peripheral's status, such as:

1. The peripheral is READY (or NOT BUSY).
2. The peripheral *fails to test* satisfactorily.
3. The peripheral has interrupted the computer, etc.

The block diagram of a typical PCU is shown in Fig. 5-11. When commanded by the computer, it accepts data, and in itself generates a punch strobe that causes the punch to accept data to its own buffer register. The computer, in turn, can interrupt the READY /BUSY flip-flop, and the program will detect when a new character is able to be clocked to the PCU data buffer.

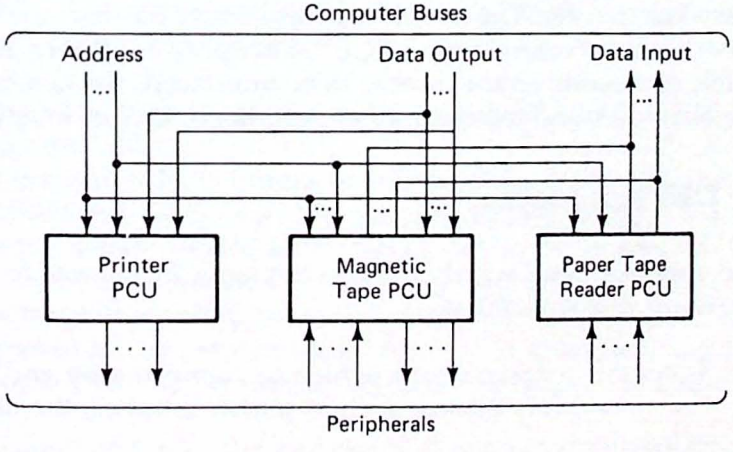


FIGURE 5-10. Input/output channel and interfacing.

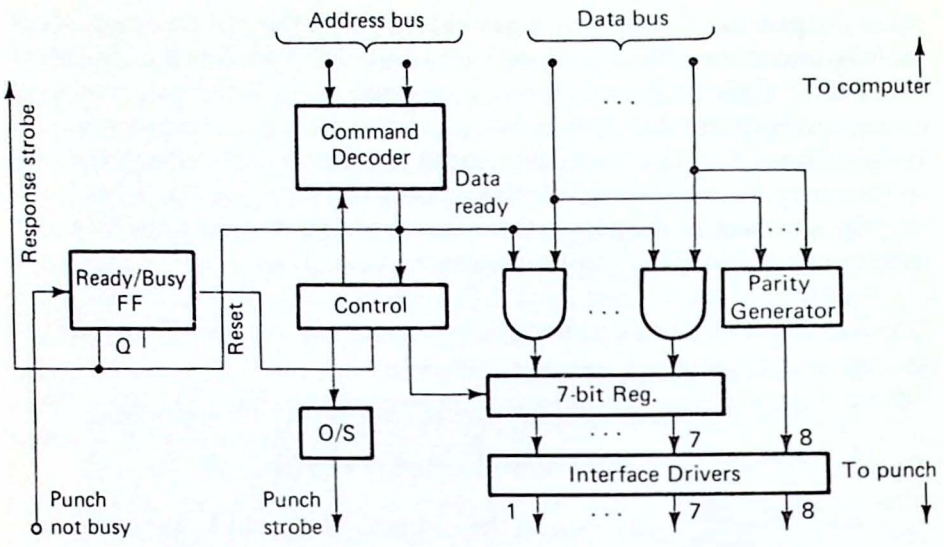


FIGURE 5-11. Typical PCU for paper tape punch.

5-9 METHODS OF INTERFACING COMPUTERS AND PERIPHERALS (H316)

Because computers and peripherals differ widely in functional speeds and concept, the general approach over the years has been to provide logic entities that perform the mating functions, such as to establish data transfer paths between the processor and a particular peripheral to which (or from which) processor-peripheral communication is required, to provide electrical interface to (from) cables, to provide data paths, to provide timing and a means to recognize a reliable (or unreliable) data transfer. The interfacing logic entity has become known either as a peripheral control unit, a PCU, or simply a controller. Designs vary widely depending on the devices to be interfaced, the general time frame of the design, and the quantity of peripheral devices involved.

5-9.1 DIO Approach

The approach most widely used in the early PCUs was to output one or more control words that:

1. Select the peripheral with which to communicate and, in the case of two-way devices such as magnetic tapes, the direction of transfer.
2. Move from (to) the memory cell address to where a transfer is to occur.

3. Check the readiness of the peripheral for data transfer.
4. Place the data in position to transfer (on input or output bus).
5. Complete the transfer by either:
   a. Strobing data to a register in the PCU (COMPUTER-OUT TRANSFERS).
   b. Strobing data into a computer register (COMPUTER-IN TRANSFERS), and verify that a proper transfer has, in fact, occurred.
6. Release any common path so that it may be reassigned for a new transfer, perhaps from a different PCU.

Sample input /output (I /O) commands in representative mnemonic form have distinct functions, as follows for the H316:

SKS Commands a particular peripheral device. For data transfers establishes a direct path between the memory and the peripheral. The next PIN (or POT) command transfers data between the peripheral addressed by the last SKS command and the memory address of the PIN (or POT) command.

PIN A command that transfers data from the peripheral last addressed by an SKS command to the memory cell (addressed by PIN).

POT A command that transfers data from the memory cell addressed by POT to the peripheral addressed by the last SKS command.

SKN This command interprets a code on the INPUT bus as discrete input signals.

SKE This command interprets a code on the output bus as discrete output control signals.

These commands are typical I/O commands for the HDC601 military processor also.

In the earlier PCU interfaces (for example, HDC 601), the address and function code of SKS was transmitted by instruction code to specify to (or from) which device a data word was to be transferred. The device (PCU) would store this function code and address. The next memory data transfer would be to (or from) this particular PCU via a POT or PIN command. The transmission of a new PCU address and function code would clear all previously specified paths and reestablish a new path.

In this approach, the peripheral command code was inserted on the output bus (and at the same time, a strobe generated in order to identify the coded bits as a command). The addressed PCU then de-

coded a POT or PIN command, which would allow the next grouping of bits on the OUTPUT BUS (or INPUT BUS if the data transfer was to be inward) to be interpreted as data. Both POT and PIN directly addressed memory.

Another interfacing approach typified by the Computer Control Corporation (Later a division of Honeywell) models DPP416, DDP516, H316, etc., is to provide a triple set of buses. We still have the OUTPUT and INPUT buses, but in addition there is an ADDRESS bus. The ADDRESS and function code to the PCU were provided via the ADDRESS bus. Data words IN and OUT are via the INPUT and OUTPUT buses, respectively. In this type of interface, the ADDRESS bus and INPUT (or OUTPUT) bus are active simultaneously during a data transfer.

Address Bus (ADB) This bus (10 bits) applies an address and function code to the peripheral PCU.

Output Bus (OTB) This bus (16 bits) provides the path for outgoing data from the processor.

Input Bus (INB) This bus (16 bits) provides the path for incoming data to the processor.

Outgoing Strobe Lines

OCPLS This strobe identifies an OCP command, which is a direct command to the PCU and does not require a PCU response.

RRLIN This strobe is employed on OTA commands to strobe data to the PCU input register. On an INA command, it clears the PCU output register after a data inward transfer. It is not used for an SKS command.

SMXO1 This strobe sets a flip-flop that unmasks an "interrupt" to the processor from a particular PCU.

CMXO1 This strobe resets a flip-flop which masks off an interrupt to the processor from a particular PCU.

In the interface, the ADB holds the PCU address and function code for the entire instruction time during any I/O instructions. During an OCP command, the OCPLS strobe is issued, which is used in the PCU to identify positively an OCP instruction.

The OTA and INA instructions are not identified by a particular strobe. RRLIN is issued by the computer in both cases and may be used by the PCU as follows:

OTA Strobe data from the processor OTB into the PCU.

INA Reset a register in the PCU inputting data on the processor INB.

An input strobe, DRLIN, signifies to the computer a successful INA or OTA transfer, or that an SKS query command was accepted.

In some of the newer minicomputers, bidirectional buses are employed; that is, data in both the inward or outward directions employ the same wires. This is possible because of special line-driver /receiver logic (such as National Semiconductor Corp. 8T10), that can both transmit data and (when the transmitter's low impedance is biased off) receive data. Also, the address bus can be employed not only to address a peripheral, but also to respond to a query which transmits inward to the processor the address of the peripheral PCU queried. Computers such as the Honeywell H716, the Prime 200, μ-Processors and others use this form of I /O bus.

5-10 DMA/DIO CONTROL

DIO control has just been described. The main difference nowadays is that buses and data paths are bidirectional. Also DIO and DMA are selected by computer commands, and the buses and control strobe are generated as needed for the different types of control. DMA transfers data directly to /from memory at a rate set by the peripheral involved. The computer finishes its present instruction and then changes over to the DMA mode of control. External devices are priority rated, and the transfer takes place to the highest-priority device signaling for service. The initial address to which the transfer between memory and the device is to take place is selected by the program and stored in the DMA channel control. Other schemes allow the peripheral to inject the initial address. A final address must also be selected and stored. When each word is transferred, the present address is incremented by 1, starting with the initial address. DMA systems store transfer addresses in registers. (DMC in Chapter 10 uses memory storage.) The present address is compared to the final address. If there is no comparison, the process continues. When they compare, the DMA process is completed and the system drops to a lower-priority channel (channels are interwoven) or back to the former program being executed before DMA was called for. Strobes are provided that request and provide a response as for DIO. Figure 5-12 represents a bidirectional DIO /DMA channel.

5-11 SUMMARY

This chapter summarizes the functions of various sections of a binary digital computer of the general Honeywell DDP 516 or H316 types. It discusses, in a general way, memory (including core stacks), required electronics and semiconductor memory forms, CPU processing opera-

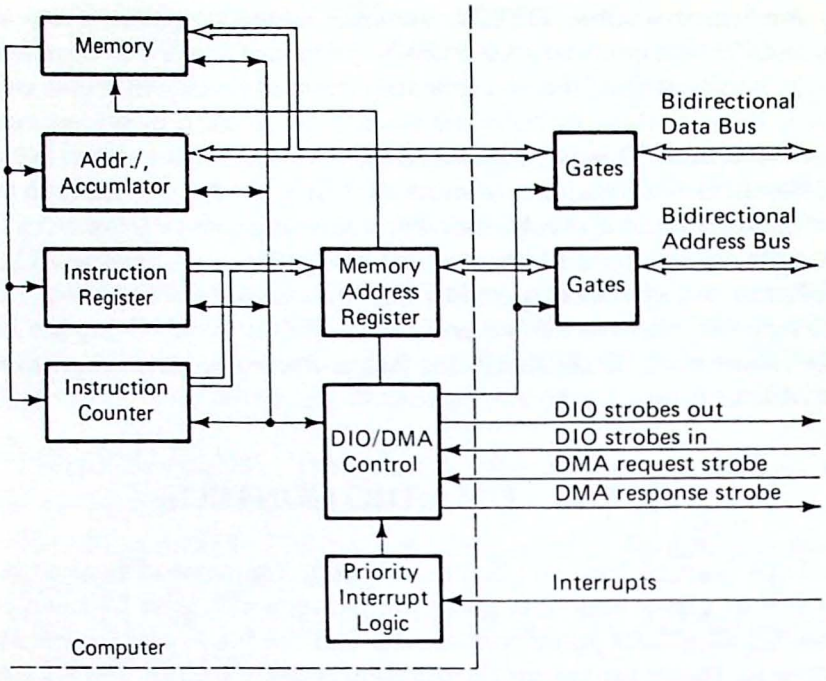


FIGURE 5-12. Bidirectional DMA/DIO interface control block diagram.

tions, double-precision and floating-point operations, and input /output methods. It points out generally that all processors are not necessarily binary coded, but can be designed to operate with a variety of codes, including BCD, octal, or hexadecimal. Minicomputers favor a binary based organization, and are most often octal or hexadecimal coded.

5-12 PROBLEMS

1. Provide some reasons why semiconductor memories have replaced core.
2. List the functions of the : (1) memory address register, (2) buffer, (3) instruction counter, and (4) index register.
3. What is floating-point operation? Why is it important?
4. Write 500,000 in floating-point format (assume that the sign is +).
5. If the range of exponent is −20 to +20, what is the typical floating-point format for+.5031?
6. In the floating-point format above add+.5031 to+301.2.
7. a. In floating-point notation subtract+80 from+1032.

b. Divide + 1032 by + 80.

c. Multiply + .503 × + 301.

8. a. What are the steps to decode an instruction?

b. What is indexing?

9. What are techniques to extend addressing capability?

10. Describe for the PCU of Chapter 5 the functions of (a) the address bus, (b) the input /output (I /O) bus, (c) decoding, and (d) test of peripheral status.

6

Peripherals

6-1 GENERAL

Peripheral equipment plays an important part in the convenience and efficiency of any processing installation. This fact is especially true in a business or commercial environment, since most of the data handling can be accomplished by relatively standard equipments that are serviced by operators of the data-processing center. Since the operators continually manipulate, adjust, load, unload, and service the equipment, they often judge the total system performance on their experience with the peripherals. This chapter discusses briefly the peripherals associated primarily with the business processor but also usable with other classes of processors as well.

6-2 LOADING AND UNLOADING DEVICES

The equipment for both loading and unloading programs and data respectively include paper tape equipment, IBM card equipment, some forms of magnetic tape equipment, and special devices such as character recognition equipment, often called *document readers*. Several of these equipments will be discussed briefly.

6-2.1 Paper Tape Devices

Paper tapes having data codes appearing as combinations of hole patterns represent a large porportion of data loading /unloading equipment. The usual arrangement is to represent a character one at a time by a pattern that may have from four- to eight-bit position columns vertical to the tape edge. The most common code these days is USASCII, employing seven columns for the code and an eighth column to store a par-

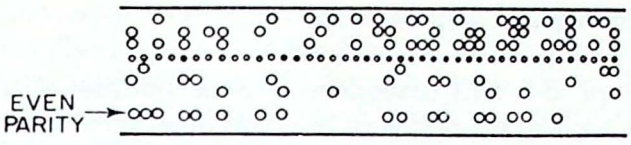


FIGURE 6-1. Example of paper tape.

ity bit. Figure 6-1 shows a sample of tape coded in this manner. Characters are spaced at 10 /inch of tape.

Paper tapes are read by devices called *paper tape readers*. The principle of operation is, of course, to sense the hole pattern and to decode the results. Slow-speed paper tape readers (10 characters /sec to 100 characters /sec) generally use a mechanical method to sense the holes (wire brushes, star wheels, etc.), while faster units (100–500 characters /sec) sense the hole pattern photoelectrically. Tape is held most often by reels, while mechanisms of many sorts both control the tension across the reading surface and provide tape slack to prevent tearing of tape during its rapid trip to and from the reels. Readers designed to handle only short tapes often dispense with the reels and allow the tape after reading to fall to the floor.

Paper tape punches employ separate solenoid-operated punches for each track. The punch mechanism pushes a pin through the paper at the desired time and in the desired pattern to record a character. These devices also vary in speeds from 10 characters /sec to well over 200 characters /sec. Reels for storing tape are commonly used with punches as well as readers.

6-2.2 Card Devices

Hollerith (IBM) cards also serve as loading and unloading devices. In this case the cards are loaded via an input hopper and are collected after use in an output hopper. They are routed through the reading or punching position one card at a time, traveling in the direction parallel to the long side. On a typical card 80 characters can be stored in the 80 columns. Other storage forms are possible, but the character method is the most common. The code employed is shown in Fig. 1-2 and is called the *Hollerith code*. Note that it differs in form from the USASCII code, and often there is a translation process required within the processor to make card-stored data and tape-stored data compatible. Card punches operate up to 300 cards/min, while reading rates can exceed 1200 card/min. Card equipment puches holes mechanically (as for paper tape) and reads either mechanically, photoelectrically, or electrically by wire brushes in the same manner as for paper tape.

6-2.3 Incremental Tapes

Paper tape and card equipment, whose traditional function has been the loading and unloading of program data, now face a new competitor in the form of incremental magnetic tape. This new equipment reads and writes at speeds competitive to the paper tape devices, but in addition stores information much more densely (from 200 to 800 characters /in.). Besides this, it is usually lower in cost. A typical unit, which appears in Fig. 6-2, can both read and write; hence, it provides both functions of the paper tape reader and paper tape punch in one device.

The incremental magnetic tape transport differs from the conventional magnetic tape transport in that the tape motion stops after reading or recording a single character. This means that during the data-store mode data may be recorded at unequal intervals of time, but the spacing on the tape is constant. Also, the standard IBM formats for magnetic tape may be retained. For conventional continuous-run magnetic tape (CRMT) transports, see Sec. 6-4.2.

Incremented tapes can be read on conventional magnetic tape transports of the continuous-run category.

6-3 DISPLAY DEVICES

6-3.1 Printers

Low-speed Printers. Alphanumeric forms of printout are probably the most common form of processor output. Nowhere in the peripheral field are there more variations than among printers. As an example,

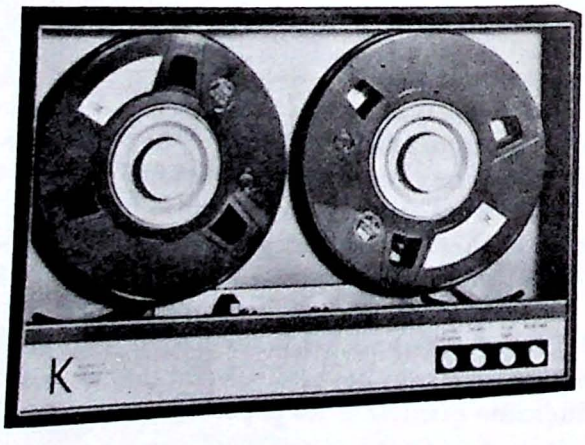


FIGURE 6-2. Incremental magnetic tape. (Courtesy of Kennedy Company.)

there is the processor typewriter that is essentially a typewriter modified to accept from the processor alphanumeric codes that actuate the letter key mechanism via decoders and solenoids. The keyboard also can serve as an input. However, the typewriter form of input /output is used only with the most minimum installations because of the relatively slow character speeds (10–30 characters /sec).

However, a processor-connected typewriter is often part of the control console and is employed here to print diagnostic or log typed messages to the operators or to serve as an operator input. Typewriters and teletype printer /keyboards serve the same functions. Quite often they are equipped also for slow-speed paper tape handling—paper tape readers and paper tape punches that also operate at the same speed and in parallel with the print operation.

Another minimum form of printer is the *strip printer*. This device prints a single strip of alphanumeric characters in much the same way as the old-fashioned telegram was put together. It also operates at relatively low speeds—10 characters /sec or thereabout.

High-speed Printers. The high-speed printer is typified by the so-called *line printer*. This device prints a complete line at a time (80–130 characters), seemingly at once. It will print typically at 20 lines /sec and in some cases faster.

The mechanism is basically a drum containing all the character fonts on a single wheel for each column. Thus for a 120-column printer there would be 120 character wheels. All characters for the line to be printed are stored as codes in a character buffer within the printer. The 120 hammers selectively tap on the paper when the selected character code for each column appears underneath the hammer as the wheels (now called a drum) rotate. Each character has a matching code that is selectively compared with the code stored in the column character buffer as the drum rotates. Note in a particular line that all A's are printed, followed by all B's, followed by all the C's, and so on. The drum rotates once for each line. After the line is printed, the paper is moved up one line position to await the hammering out of the next line during the next rotation of the character drum. Figure 6-3 illustrates generally the concept for this form of mechanical printing.

Other types of printer mechanization are receiving increased attention.

A somewhat different approach is to use thermal printing (apply heat or actually strike an arc) from a print head matrix to heat-sensitive or electroconducting paper. Usually the font is not so pleasing to the eye, but the machines operate quietly, are much less expensive, and are easier to maintain than the mechanical printout type. Dot printing is shown in Fig. 6-4. Printers using this method are popular for minicomputers. Figure 6-5 shows such a printer marketed by Perkins/Elmer Data Systems.

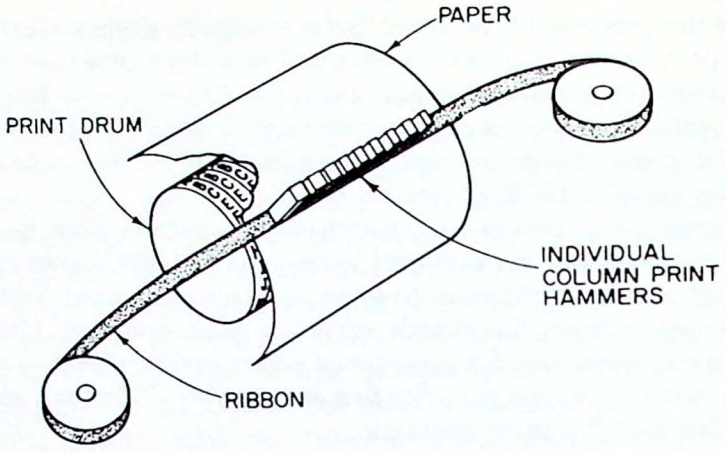


FIGURE 6-3. Drum printing mechanism.

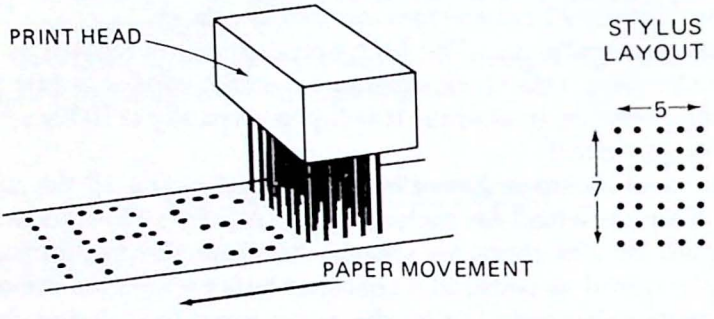


FIGURE 6-4. Dot printing (electrostatic or thermal).

6-3.2 CRT Displays

Where quick looks are desirable and where no permenant record need be kept, alphanumeric displays as shown in Fig. 6-6 provide a convenient way of presenting information. This equipment's keyboard also serves as an input device and provides a convenient means for the operator to interact with the computer to edit, display, and observe changes that he is making to the program by his editing process.

6-3.3 Plotters

Functionally these devices are similar in use to CRT displays but can provide a permenant readout of variable display data. A pen is positioned in X and Y coordinates and is moved to draw a graph or chart. In some cases high-speed printers can prepare rough charts by printing Xs in selected patterns. A device manufactured by Gould has served both as a printer and plotter.

FIGURE 6-5. Matrix printer. (Courtesy of Perkins-Elmer Data Systems, Tinton Falls, New Jersey.)

6-4 BULK STORE MEMORY PERIPHERALS

6-4.1 General

Because the internal memory of a processor has an upper limit to its size—usually by design or cost consideration—systems that must store large amounts of data external to the processor require an economical storage facility. This is the function of the bulk store peripherals. They are true peripheral devices, because their interface is via PCUs connected to the *input /output* of the processor. The important characteristics of the bulk storage devices are their storage capabilities, the cost per bit of storage, and access time. *Access time* is defined as the time between the request for data and the actual start of the data transfer.

Where access can be in the order of seconds or minutes, the *continuous-run magnetic tape transports* are favored. Their cost per bit of storage is the lowest readily available, and their storage capability is great.

If access times in the order of milliseconds is desired, *magnetic drums or head-per-track disk files* are generally specified. Here the cost per

FIGURE 6-6. CRT display. (Courtesy of Sperry-Rand UNIVAC Division.)

bit of storage is about ten times that of magnetic tape. Intermediate between the two equipments just described is the disk file having movable read /write heads. These heads must be positioned before the data transfer begins hence *access time* on the average is greater than for the *drum in head-per-track disk file*. The longer access is, of course, caused by the head-arm positioning requirement. These letter disk files are constructed so that the disk packs are removable; hence they exhibit the replaceability convenience of the magnetic tapes. Table 6-2 lists for comparison purposes the relative features of each type of bulk storage.

Bulk-storage devices generally provide on-line storage to complement the machine's internal storage. The continuous-run magnetic tapes, magnetic drums, and disk files and some of the newer film storage equipment, hologram storage, etc. eventually will belong to this category of equipment. Constant development in this area appears probable in the incessant search to develop less costly, faster, and more compact forms of data storage.

6-4.2 Continuous-run Magnetic Tapes (CRMT)

This form of bulk store has been around for many years and is still the least costly on a per-bit basis. It is typified by reels of magnetic tape in which characters are recorded as code groups across the tape in a simi-

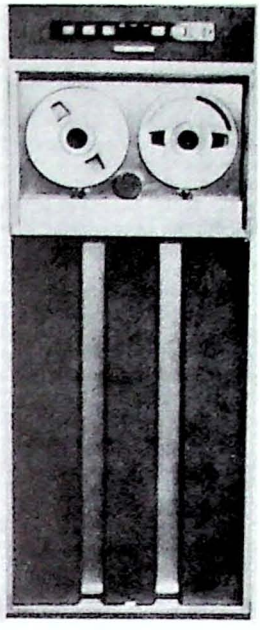


Figure 6-7. Continuous-run magnetic-tape equipment. (Courtesy of Honeywell Inc., EDP Division.)

TABLE 6-1 IBM standard seven-channel magnetic-tape formats and specifications *

1. Number of tracks	#7 (6 data, 1 check bit)
2. Character density	#200, 556, 800 bit /in.
3. Data format	Binary or character
4. Type of recording	NRIZ † (Ferranti) two-gap heads (write-read)
5. Tape speed	75 or 112.5 in. /sec; rewind, 500 in. /sec
6. Error check	Vertical: odd parity for each character Horizontal: odd parity for record of each track Check of character (LRCC ‡) recorded in EOR gap
7. Groupings	Record: group of words ending in 3 /4-in. wide gap (EOR) File: group of records ending in 3½-in. wide gap and /or tape mark character (EOF)
8. End-of-tape mark	3 /16 by 1-in. aluminum strip, 10 ft from tape end, sensed to stop unit
9. Tape-capstan start-stop time	Less than 3 msec

Notes to Table
**New USASC II standards allow 9 tracks and 6250 bits/in. spacing as well.*
†NRLZ–designation for Ferranti or Manchester recording code.
‡LRCC-designation of parity check character recorded in End of Record (EOR) gap.

lar format to paper tape, but the magnetic spots replace the holes (and no holes). Figure 6-7 shows a CRMT tape drive unit. In general, the industry has standardized on the IBM format (Table 6-1). The unique characteristic of the CRMT is that data are recorded in blocks called *records*. Groups of records are called *files*. Data transfers to and from the processor are restricted to *records* and *files*.

In addition to LRCC (a logic redundancy check code) CRCC is recorded before LRCC. The interrecord gap now becomes 0.6 in. Between successive records (or files)there are distincitive gaps in the recordings. These are necessary, since the machine reads and records with the tape moving continuously—not intermittently, like incremental magnetic tape devices. In addition, control characters to signify *end-of-file* (EOF) and *end-of-record* (EOR) are recorded. The gaps serve to allow the tape to run up in speed or run down in speed between records, where the tape motion ceases. The tape transport mechanism is characterized by reels to hold the used and unused tapes and to provide the necessary tape feed and takeup. In addition, there is a capstan to drive the tape at a constant speed across the read and record heads and a mechanism to control the slack between the reels and the driver capstan.

6-4.3 Magnetic Drums

This equipment stores data about the circumference of the drum. There are numerous parallel tracks, all having their individual read / write heads. Because the surface speed is high and the surface itself is unyielding, the heads are designed to float a few microinches off surface. Otherwise, the wear to the head (or surface) would be excessive and poor equipment life would result.

6-4.4 Disk Files

Disk files perform a similar function to drums but are organized with the data tracks coaxial. One form of disk file has a head for every track and is thus functionally identical with a drum. Another form of disk file has a bank of heads that each read or write one track at a time concurrently on eight concentric disks. Thus an individual character code may be spread among several disks but will have a common corresponding track position on each disk. The arm holding the heads is positioned by a servo under control positioning instructions from the processor. In addition, the stack of disks is usually replaceable, which allows the operator to replace disk packs in the same manner as tapes are replaced during their use. Figure 6-8 shows this latter type of disk file.

FIGURE 6-8. A typical commercial disk file. (Courtesy of IBM Corp.)

6-4.5 Comparison of the Magnetic Bulk Stores

A gross comparison is accomplished by Table 6-2. Generally magnetic tapes store data in a less costly form, but the access to a given list of data is rather slow. There is usually no addressing of the records, although it can be done by starting each record with a recognizable code that a program instruction can compare with a reference.

Drums and separate head-per-track disks are functionally identical and have the fastest access time of any magnetic-mechanical form of storage. The individual tracks and data circumferential position on the track can be addressed. Usually the data are transferred in record groups, or file groups.

As pointed out previously, the other form of disk file utilizes head groups that are servo positioned to a selected track. Again, the positioning is addressable, and the transfer is in record groups. However, this form of disk file is slower to access than the head-per-track type due to the requirement of first positioning the heads of the latter. Access, however, is normally faster than for magnetic tape and has the advantage over the fixed-head disk in the replaceability of the disk packs themselves. This feature reacquires one of the advantages of replaceable

TABLE 6-2 Comparison of Bulk-store Devices

	Storage (bits)	*Transfer (characters/ sec)*	*Density* $(in.)^2$	*Access*	*Approximate cost ($/bit)*
1. Magnetic tape					
Continuous Run	16.1×10^7	56K	800	125 sec	0.0015
Incremental	8.1×10^7	2K	800	48 min	0.00005
Cassette	3.6×10^6	4K	180	300 sec	0.001
2. Rotating					
Drums	1.5×10^6	300K	500	8 msec	0.015
Disk file (fixed head)	5.0×10^6	300K	1000	8 msec	0.005
Disk file (movable hd.)	2.0×10^6	200K	500	200 msec	0.010
Floppy disk	3.1×10^6	600 bytes	3200	10ms/track	0.002
core					
3. Magnetic	1.6×10^6	-	-	1-5 μsec.	0.02
4. Other (hologram, laser, CCD, magnetic domains, etc.)	96×10^6		.012 ms .128 ms		

magnetic tapes. The major features of these equipments are summarized by Table 6-3.

A newly emerging form of low-cost head-per-track disk file is the so-called *floppy disks*. These are usually single flexible disks with a number of concentric tracks. These are designed to be in the price range of minicomputers, and have binary serial outputs /inputs with heads movable from track to track. A *floppy disk* is shown in Fig. 6-9. This can be single-sided or two-sided.

6-4.6 Cassettes

The cassette, so popular for music recording, has been adopted as an inexpensive plug-in form of storage. The quality of computer-operated tapes, drives, etc., is much better than that of those built for entertainment uses. These devices are excellent for loading short programs, as one would use a paper tape. A typical cassette system is shown in Figure 6-10.

TABLE 6-3 Important Features of Bulk-store Devices

A. Magnetic tape
  1. Usable for
    a. Occasionally accessed bulk store (where low cost per bit is important)
    b. Load and unload main processor memory
    c. Where tapes must be replaceable
  2. Transfer characteristics
    a. Multiword in records of files
    b. Via buffered channel (DMA)
B. Disk files
  1. Usable for
    a. Data to be accessed often
    b. Moderate cost for storage ($ /bit)
    c. Some forms of disks replaceable
  2. Transfer characteristics
    a. Multiword in records or files
    b. Via buffered channel (DMA)
C. Magnetic core
    a. Single-word access (via DIO)
    b. Relatively fast access
    c. Moderately expensive on a per-bit basis
D. Other
    a. Semiconductor—faster and less expensive than core for smaller memories
    b. CCD and magnetic domain—a few units are now available (1979)
    c. Hologram—in development

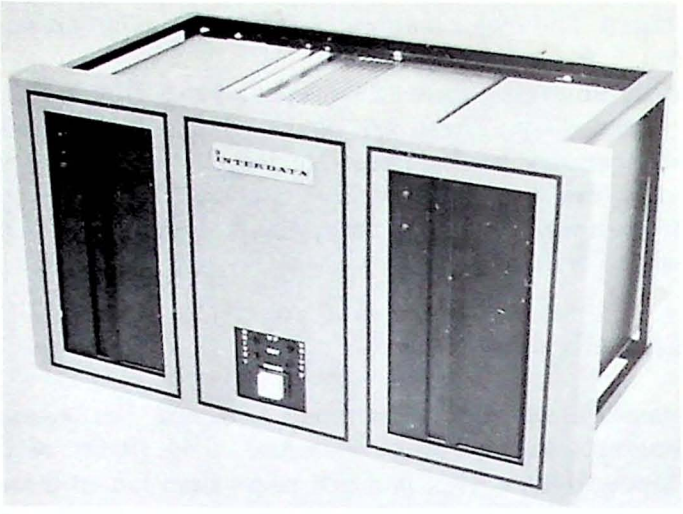


FIGURE 6-9. Floppy disk. (Courtesy of Perkins-Elmer Data Systems.)

6-4.7 Charge Coupled Devices (CCD) Bulk Store

Although disk files and magnetic tape units have traditionally provided bulk storage, some of the newly developing technologies appear to offer improved capabilities. In the early computers data storage involved rotating bits (for example, a drum computer). This technique can now be updated to include CCD shift registers, which can be ideally suited to this technique, since they are temperature insensitive (as mercury delay lines were not). CCDs essentially store electrons (or their absence) in potential wells—regions near the semiconductor surface. Once the charge is stored, it can be transported around by applying voltages between adjacent electrodes. Thus, as for a logic shift-register (1s) and (0s) can be shifted from the first cell to the final cell in successive steps. The storage process continues by tying the output to the input (recirculation) and reading an output from the last stage. The CCD can be designed so that many parallel data streams can be recirculated at the same time. In this way a complete character code or word, or groups of words, can be shifted at each clock time if sufficient parallel tracks are provided.

Advantages to this type of storage over mechanical storage should be self-evident, since there are no large rotating disks with their inherent mechanical movements. These units are compact—well over 200,000 bits per square inch. Recirculation time is at the machine's clock rate. These are finally being offered from suppliers in small quantities, but research is continuing.

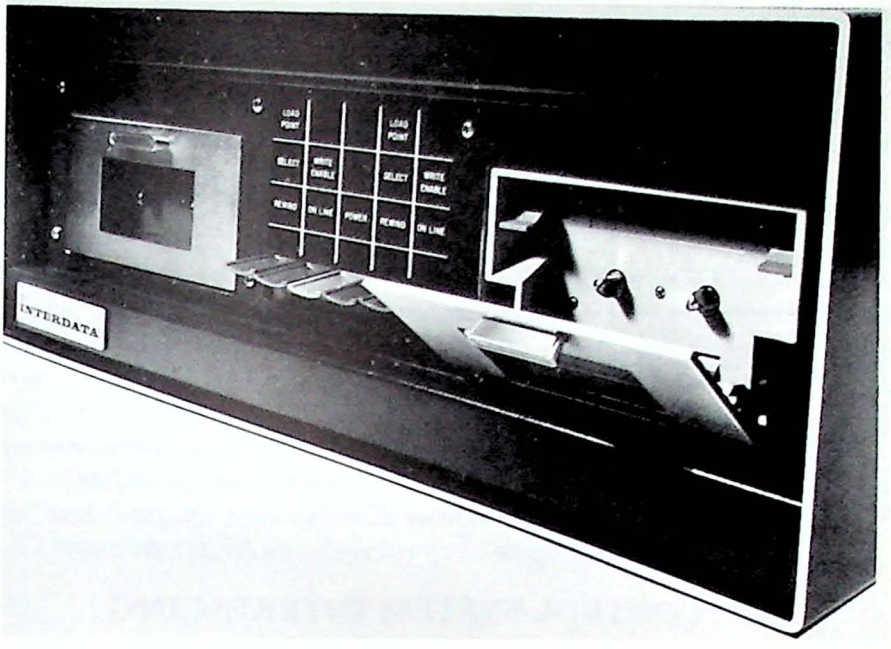
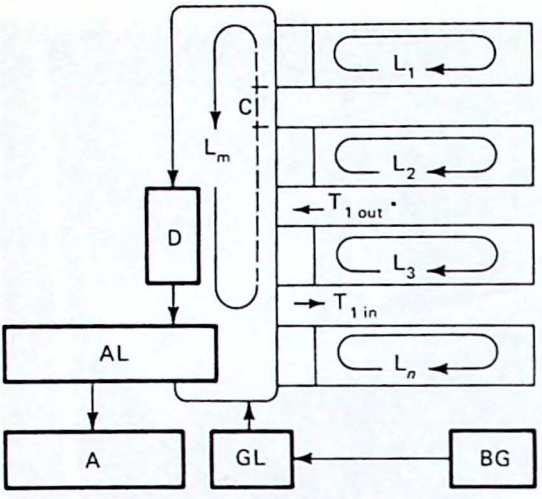


FIGURE 6-10. Cassette storage. (Courtesy of Perkins-Elmer Data Systems.)

6-4.8 Magnetic Domain Bulk Store

A new 92K-bit magnetic bubble memory has been introduced by Texas Instruments. Operation of a bubble memory can be viewed in Fig. 6-11. Bits to be stored (bubbles) originate in the bubble generator and are passed via a one-way generator lock to the major loop. Bit distance p determines the maximum number of serial bits in the major loop between suscessive and adjacent minor loops (L_1 to L_n). Minor loops permit circulation in one direction only controllable by a rotating magnetic field. Reversing magnetic fields at the proper time permits switching word strings in or out of a minor loop to the major loop. Data readout is via the annihilator lock of the bubble annihilator. Access time for any word can take several milliseconds. 92K-bit memories are now available, with 1-million-bit memories expected soon. There are 661 bubble positions in each of 157 minor loops (100,657 bits), but allowance must be made for 8 percent defective loops, reducing the 157 figure to 144, yielding about 92K bits. The cycle for the 144-bit page is 12.8 ms; thus it is expected that bubble memories will soon compete with disks and drums—along with CCD memories, which at this point are 1000 times as fast as bubble memories.



BG	Bubble generator
GL	Generator lock
A	Annihilator
AL	Annihilator lock
D	Detector
L_m	Major loop
$L_1 \rightarrow L_n$	Minor loops
p	Bit distance
$T_{1\ in}$	Input switch for minor loop
$T_{1\ out}$	Output switch for minor loop

FIGURE 6-11. Functioning of a bubble memory.

6-5 CONTROL SYSTEM INTERFACING

6-5.1 A/D and D/A Converters

6-5.1.1 A /D Converters Often in a control system, many analog signals must be sampled and fed to the computer for their individual evaluation. This approach requires, first of all, a multiplexing device (sort of a controlled input switch that samples each analog signal in its proper term and converts these inputs in turn to a single line). This process of multiplexing is controlled by the computer program. On the output side of the multiplexer (input to the computer) is an analog-to-digital (A /D) converter. This device reads each analog signal multiplexed to it, and develops a corresponding digital number. This number, now in a format understandable to a digital computer, is inputted via an input channel as if it were any peripheral. Typical A /D converter binary words are from eight to fifteen bits long, or can be in BCD code.

6-5.1.2 D /A Converters Many devices connected to digital control systems can be analog in nature. Since the output of any digital computer is a binary (or coded) word, a device is needed to restore a digital signal to analog form. This is the object of the digital-to-analog (D /A) converter. It is possible to connect the analog output to a demultiplexing device (a device that supplies inputs to many analog channels from a single digital output. Again, which device is to receive the output is determined by the computer program). In both cases, A /D and D /A

conversion special analog circuiting (called normalizers) are required to standardize input and output analog signals to a nominal range within the converter's capability (for example, ±10 volts dc).

6-5.1.3 Encoders (used for shaft-type inputs and outputs) If the analog signal to be handled by a computer is a shaft-rotation, there are alternatives to A /D and D /A converters. These devices are called *encoders*. Depending on the exact position of the shaft, a particular digital code is sent to the computer input channel. On the output side, a digital word can be transmitted to a register, and by means of a feedback servo, an encoder can be rotated by a servo motor until its code matches that of the register. Often today, the servos are electronic devices rather than gear trains and motors; however, if the output is desired as a shaft position, electromechanical servos are still in vogue.

The aforementioned *A /D, D /A converters, multiplexers, demultiplexers,* and *encoders* are common to control real-time applications. Figure 6-12 illustrates in block diagram form these devices.

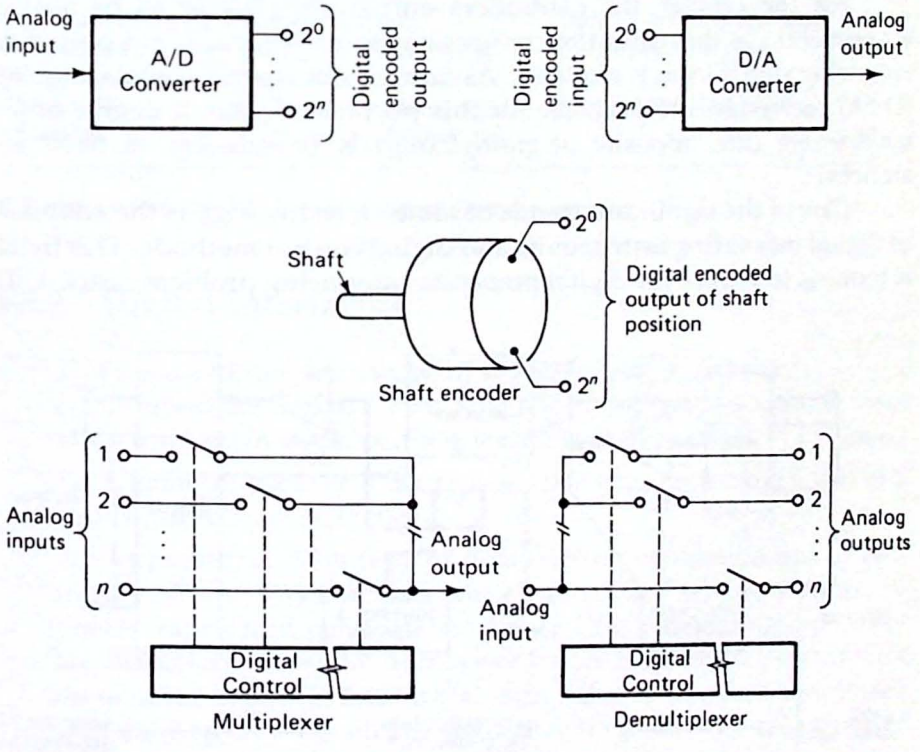


FIGURE 6-12. A/D, D/A converter, digital encoder, multiplexer, demultiplexer block diagram.

6-5.2 Analog Interfacing

In this class are those devices that interface the processor to measuring instruments of all types (usually inputs to the processor) and provide output to various controllers. Usually human intervention is not feasible or desired in these systems that in many instances are examples of automatic control.

If the measuring instruments are analog in operation, then an analog to digital conversion is required before the measurement data is in a form a digital processor can digest. Analog to digital (abbreviated A /D) conversion requires the use of an A /D converter (a hardware device that performs the conversion process at a sufficiently high rate to capture all significant variations in the variable being measured by the analog sensor). Since many measurement parameters vary slowly with time—such as air pressure, temperature, and flow rate—it is usually possible that several sensors can share a single A /D converter, provided that the sensors are switched periodically and sufficiently often into the A /D input. A *multiplexing switch* will accomplish this selective switching process.

For the output, the controllers employed a likely to be analog equipments. In this case, the conversion from a digital word output to an analog signal form is required. As one would expect, digital to analog (D /A) converters are available for this purpose. Again, a degree of *demultiplexing* (the opposite of *multiplexing*) is permissible in most instances.

One of the significant trends of modern technology is the adoption of digital measuring instruments and digital control methods. This trend is tending to reduce the digital processor interfacing problem, since A /D

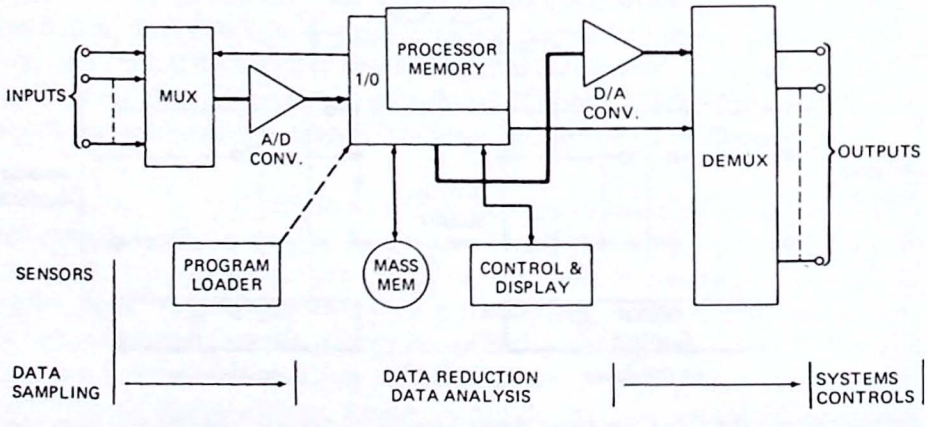


Fig. 1-11. Real-time system with converter interfacing.

FIGURE 6-13. Real-time system with converter interfacing.

and D /A conversions should not be required as often in the future. A /D and D /A converters, along with the switching devices, tend to be expensive; hence, the all-digital approach promises to be less costly on a systemwide basis. Presently it is difficult to completely eliminate the analog conversion process, since many control equipments employ such devices as *synchros*—which by their nature are an analog device requiring three (possibly two) A /C analog inputs or outputs depending on their use in the system. Although these can be replaced by digital shaft encoders, synchrolike devices that sense a remote shaft position, many manufacturers are reluctant to make the change in their designs.

A general diagram for a real-time system is shown in Fig. 6-13. A brief tabulation of the characteristics of a number of the real-time devices discussed appears in Table 6-4.

Table 6-4 Real-time Device Comparisons

Device	*Function*	*Speed*
A /D Converters *	Analog inputs	Up to 1 mHz conversions /sec
D /A Converters *	Analog outputs	Up to 1 mHz conversions /sec
Multiplexers / Demultiplexers	Combine channels at input and output	Up to 500 kHz /sec
Digital I /O Devices	Buffering, switching, formatting, control	Computer I /O speed

**The ac synchro signals are all interfaced via these devices.*

6-5.3 Digital Interfacing

This form of interface can be of several types, as follows:

1. A complete digital word, which for a control processor is most often from eight to sixteen bits transferred in parallel.

2. Character parallel. This type of data transfer is from 6 to 8 bits and is normally ASCII code.

3. Incremental. This type of data transfer is often on one of two transfer lines. Positive data are characterized by the number of pulses transferred per unit time on one line, whereas negative data are characterized by the number of pulses transferred per unit time on another separate line. If the data value is (0), then the pulses will alternate equally on the two lines. If the data value is tending toward a positive value, there will be more pulses on the positive line relative to the negative line. This, of course, is reversed for the slightly negative-going data.

6-6 SUMMARY

This chapter in a very general way summarizes peripheral devices. The classes defined include:

1. Loading or unloading a computer's memory.
2. Those that interface with people.
3. Those that interface with bulk store devices (disk, drums, etc.).
4. Those that interface with control systems; that is, A /D and D /A converters.

These devices are changing so rapidly that it is almost impossible to stay abreast of the field. With the cost of mainframes dropping rapidly, it is expected the costs of peripherals will drop also; in fact, this has been happening at a phenomenal rate.

6-7 PROBLEMS

1. Name at least three loading and unloading devices. Are they loaders or unloaders?
2. What devices can an incremental tape unit replace?
3. Cathode ray tube devices (CRTs) permit the operator to do two principal things to alphanumeric data. What are they?
4. What are the main advantages and disadvantages to CRMTs?
5. How do high-speed printers differ primarily from low-speed printers?
6. Magnetic drums and disk files, although mechanically different, accomplish the same general task. (a) What is the task? (b) Why is access to a disk or drum faster on the average than to a CRMT?
7. What are the advantages and disadvantages of a floppy disk over a conventional moving-arm disk?
8. If the head on a floppy disk were to move two tracks, what would be the average access time to 8 bytes? (Use Table 6-2; assume that a byte is eight bits, that the RPM is 300, and that the bits are packed 1600 to the inch.)
9. What are A /D and D /A converters?
10. If the input from digital devices is via multiplexor, what is the sample time from one device if it takes 50 μsec to step the multiplexor through five devices.

7

Instructions and Programming

7-1 INTRODUCTION

The program of any computer is the sequence of instructions (also called *commands*) that are loaded into its internal CPU memory previous to operation. The computer interprets each of the instructions in turn (sequenced by the instruction counter), and thereby executes the program. In all but the simplest machines the sequence of instruction execution may be varied by inserting a new and different program. The term *programming* generally refers to the entire complex process of breaking the problem to be solved down to elementary steps that the computer can perform, writing down the instruction steps in sequence, converting a *source* program to an *object* program (from man to machine language), and testing the *machine language* with representative sets of data. In other words, programming is the process of directing the machine through the desired process or sets of calculations.

A computer operates from coded (1) and (0) information. The data are coded; the instructions are coded. When an instruction is in its (1) and (0) form, it is termed a *machine coded* instruction. It was probably originally written in a *mnemonic* (English language form) and converted via an *assembler* or *compiler program to the machine code.*

In the early days instructions were coded directly in machine codes. As a result programmers knew what each code stood for. These days it is the fashion to write programs in a near-English form, and have these converted to machine codes by *assembler* or *compiler* programs, as just pointed out. The near-English program form is termed a *source* program, while the machine form is an *object* program. These processes are highly sophisticated and require extensive translation programs of their own. The computers themselves must have sufficiently large memories and the proper instruction repertoire to handle the translation process. *Assembler* programs typically convert one *source* statement to one

machine instruction, or *object* instruction. *Compilers* such as FORTRAN, ALOGOL, COBOL, or PL/1 provide languages for writing *source* statements. A single statement here may generate several machine instructions, as contrasted to the one-to-one relationship of an *assembler*.

By the use of program-oriented languages, it is possible to train people who are completely uninformed as to how a computer performs, to use a computer effectively. In addition, programmers trained in a language such as FORTRAN should be able to program in FORTRAN on another computer having this *compiler* (see Chapter 9). Actually, complete compatibility between two machines having the FORTRAN *compiler* is not always assured, because minor differences exist frequently between the two machines' FORTRAN program details.

It is not unusual for computers in the same product line to use the same *assembler* language. Thus, programs written for different machines of this line will run on other machines of the line. The next section discusses instruction types. *Assembler* language programming requires that the programmer have a general understanding of how instructions perform and what he has to work with. The examples chosen are from the Honeywell H316 repertoire.

7-2 FIELDS OF INSTRUCTIONS

Groups of bits (known as fields) are examined together by computer decoding logic as a unit to specify control functions of an instruction. For example, there will be a bit group that identifies specifically the instruction's type. This field is the *operation code*, or more usually just the *op code*, and identifies exactly what control action the instruction is to perform. Is the desired action to ADD the contents of the memory cell to the accumulator? Is it to move accumulator data contents to the output bus? Is it to shift the contents of the accumulator eight places to the right? All of these operations, and many more, the *op code* field specifies. Another field in the instruction can address a cell of memory. Other specific bits in instructions may specify other control actions, as we shall presently see. Instruction discussed in the sections here and to follow are most often 16-bit words. In some cases 16 bits will not be sufficient to specify even the *op code*. In some cases the *op code* and *address* together require more than 16 bits. A special operation, called an *indirect address*, may be employed to address a new word having the total 16 bits available for either *op code* or *address*. A control bit in the original instruction signals this jump to the indirectly addressed instruction where more power is available.

7-2.1 Memory Reference Instructions

Memory reference instructions provide an *op code*, a *memory address*, and three control bits. The three control bits perform the following functions:

1. Bit 1, if a (1), requires the address to be interpreted as *indirect.*
2. Bit 2, if a (1), requires the *index register* to be summed with the instruction's present address.
3. Bit 7, if a (1), requires the address to be in the *base* section of memory; if a (0), it requires the address to be in the *present* section of memory, where the program is now executing. Bits 3–6 are *op code*, while bits 8–16 are memory cell address subject to bit 7's interpretation. If bit 1 and bit 2 are (0), the instruction is neither *indexed* or *indirect* and can be executed immediately.

Bit 7, the sector bit, defines whether an address is in the *present sector* or the *base sector.* By jumping back to the 512 words of the *base sector*, the program can by an *indirect address* be directed to any of 64 sectors of memory (each 512 words). For example, if the computer is operating presently in sector 5 (words 2049 to 2560) a sector bit (0) and a *memory address* of 100 will address word 2149. If, on the other hand, the sector bit is (1) then a *memory address* of 100 addresses word 101, provided that base sector 1 is words (1-512).

Instructions such as ADD, SUBtract, MULTiply, DIVide, AND, etc. belong to the *memory reference* catagory, if a *memory address* is required. Otherwise, they are found in the generic category of Sec. 7-2.3, where the entire 16 bits is available for either *op code* or *address*. Some *load* and *store* instructions, those that load memory from the *accumulator* or reverse the process, are in the *memory reference* category. Included also are some of the jump instructions, those that reference memory.

7-2.2 Shift Instructions

This instruction class generally causes a word in the *accumulator* to shift, either to the right or left, a specific number of bit positions. Shifts can be of a number of basic types, for example,

1. All 16 bits, or only 8 bits, to the left or right.
2. Shifts where bits pushed out of register ends are lost.
3. Shifts where bits pushed out one register end appear at the other end of the same register.

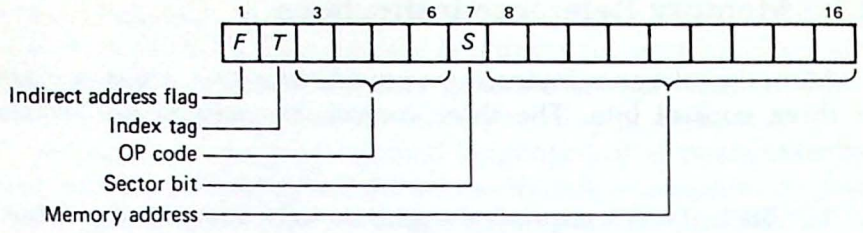


FIGURE 7-1. Memory reference instruction format.

4. Shifts where bits are pushed to other registers.

In some cases a bit is reserved to record a shift overflow—that is, a condition where a bit is added beyond the limits of the register contents—too far to the left or right. The general format of a shift class instruction is shown in Fig. 7-2. The *op code* occupies 6 bits; the number of shifts occupies a 10-bit field. The complement value for the number of shifts is inserted in the shift field.

7-2.3 Generic Instructions

This class is illustrated by Fig. 7-3, where the *op code* occupies all 16 bits. This category includes selected arithmetic operations, such as adding two registers (where no memory reference is required) complementation of the accumulator, and such operations as setting the sign of the accumulator, for example. Most of the 22 skip operations belong to this group. Other *generic instructions* include: load and store, teletype key and control panel switch commands, clear, enable interrupt, and HALT commands.

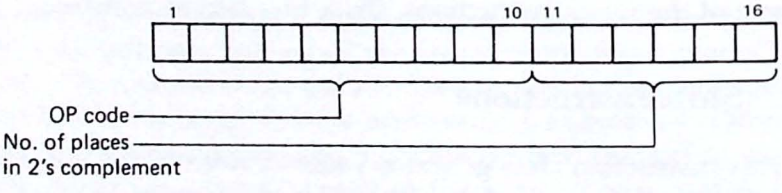


FIGURE 7-2. Format for a shift instruction.

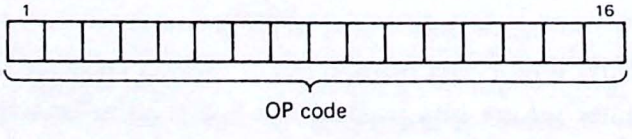


FIGURE 7-3. Generic instruction format.

7-2.4 Input/output (I/O) Instructions

This instruction class provides computer control of peripherals, such as starting and stopping or changing operating mode. Data transfers up to 16 bits at a time either in or out employ I/O commands. An *OCP Command* is an I/O command that controls the action of a peripheral. It is identified at the interface by a strobe (OCPLS) and requires no response by the peripheral to the computer. The *data output command* (OCA) allows data to flow outward to peripheral in 16-bit parallel form from the *accumulator*, while at the same time a peripheral *address* and *function code* appears on the *address bus*. Data transfers require the peripheral control unit (PCU) to return a DRLIN strobe to the computer, if it is ready for a transfer. If DRLIN is not received within the allotted time, the transfer of data is not completed, or not possible, and hence, a strobe RRLIN does not strobe data to the PCU data register. Whether DRLIN occurs or not is a signal to the program to go to the next instruction in the program (if it does not) or skip over the next instruction (if it does). Thus a programmer can program a continuous jump back until data are actually transferred. An *SKS instruction* differs from an *OTA* in that it is a query form and will not strobe data outward. It examines DRLIN to determine if the answer to the query was yes or no.

Data are transferred inward from a PCU to the computer by *command INA*. It behaves much the same way as *OTA*, requiring a DDLIN strobe input to actually transfer data. The RRLIN strobe here clears the PCU. The format for all I/O instructions is given by Fig. 7-4.

The bit structure for all I/O instructions is the same—only the *op codes* differ. During execution of *OTA* or *INA* the data are on line at the same time as the *peripheral address*.

7-2.5 Optional Equipment Instructions

Depending on which optional equipment is included within a system, optional instructions are wired to serve the optional devices. These follow standard designations and formats. These instructions operate

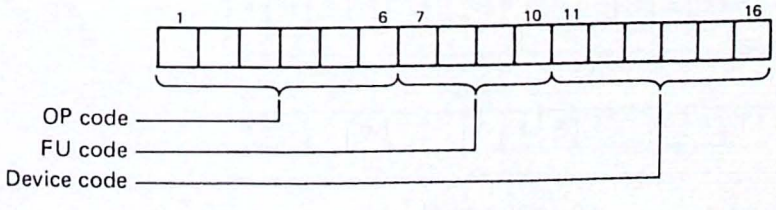


FIGURE 7-4. Formats for I/O instructions.

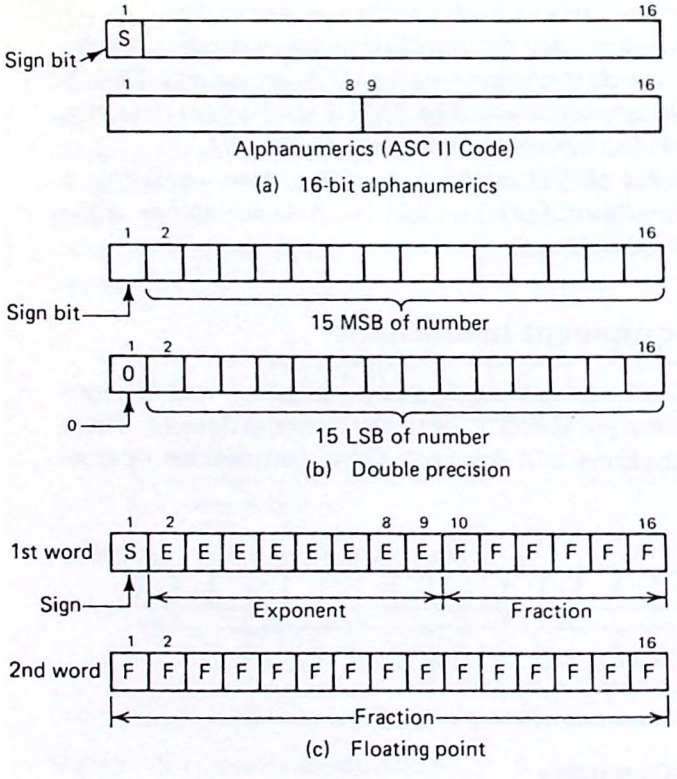
the special PCUs, which can provide extended addressing past 32K words of memory, add *memory parity*, provide *memory lockout* (where a section of memory is blocked from normal access), provide *DMA* or *DMC*, provide *priority interupts*, provide *high-speed arithmetic* operations, and provide a *real-time clock*, just to name a few.

7-2.6 Data Representation

Data may be represented by any of the forms shown in Fig. 7-5.

7-3 THE INSTRUCTION REPERTOIRE

Instructions are classified according to their primary and general functions, as described in Sec. 7-2. The exact repertoire of instructions may vary between computer models of the same general line and certainly between lines of different manufacturers. In some cases it is possible to add optional instructions and hardware that increase the speed



Two ASC II characters can be stored per word. Input and output are from bits 9-16. Bits 1-8 are loaded and unloaded via shifts from bits 9-16.

FIGURE 7-5. Formats for data words.

and capability of the machine. In a well-planned line, identical lists of instructions are employed basically, but the larger machines may have additional instructions to increase their computing or data-manipulation powers. The larger machines will be able to run programs of those further down the line without redoing the programs. This ideal situation comprises a program-compatible line (or upward and downward compatible line) and enables users to upgrade (or downgrade) systems with minimum changes to hardware or software itself. A partial list of instructions (repertoire) of the H316 computer is given in Table 7-1.

7-4 INSTRUCTION ORGANIZATION

7-4.1 Single Address Instructions

The instruction classes described in Sec. 7-2 are defined to be *single address* instructions, which means, of course, that only a single address appears in the format. The standard formats for *single address* for most

TABLE 7-1 List of Important H316 Instructions (not complete list)

Mnemonic	*Registers*	*Mnemonic*	*Registers*
1. Important load and store instructions			
CRA	Clear A	STA	Store A
IMA	Interchange Memory and A	STX	Store X
LDA	Load A	IAB	Interchange A and B
LD	Load X		
2. Arithmetic instructions			
ACA	Add carry to A	SUB	Subtract
ADD	Add	TCA	2's Compl. A
3. Shift instructions			
LGL	Logical left shift	LLR	Long left rotate involving A and B
LGR	Logical right shift	LRR	Long right rotate involving A and B
[There are also long left shift (LLL) and a long right shift (LRL).]			
4. I/O instructions			
INA	Input to A	OCP	Output control command
OTA	Output from A	SMK	Special OTA to set masks.
		SKS	Skip if *ready-line* set
5. Control instructions			
CAS	Compare	SKP	Unconditional skip
ENB	Enable interrupt	JST	Jump and store location
HLT	Halt	NOP	No operation
INH	Inhibit interrupt	SMI	Skip if A minus
JMP	Unconditional jump	SPL	Skip if A plus
SLZ	Skip if A = 0	SNZ	Skip if A not 0

control-type computers are simular to those described for Honeywell control computers in Sec. 7-2. This form of instruction is binary code, most often 16 bits, but can vary between computers from 12 to 36 bits. It performs a single operation. A typical computer can have over 100 in the repertoire.

7-4.2 Multiple Address Instructions

Another format for instructions, where more bits are available, are those containing two or more addresses within the bit patterns. These are the *multiple-address* instructions. In the EDVAC computer, back in the computer dark ages, this machine had a 44-bit code divided into a four-bit *op code*, and four addresses of 10 bits each. The first two address transferred operands to registers where the *op code* indicated the arithmetic action, or whatever, to apply to the operands together. The third address was where to store the results back in memory. The fourth address pointed to the next instruction's address. Originally there were only 12 *op codes*; hence, four more were available in the format. Whether they were ever used the author does not know.

TABLE 7-2. Types of Instruction Structures and Their Advantages

Structure	*Advantages*	*Disadvantages*
One-address instruction	1. Uncomplicated decoding	1. An individual instruction order required for every operation
	2. Relatively ssimple hardware.	2. Overall computer speed is slower.
Two-address instruction	Allows some speed-up since paralleled operation can occur, such as two words moving at the same time	More hardware than for a one-address instruction.
Three-address instruction	Permits fewer instructions, since three addresses are included. More versatile since third address can serve as a location for next instruction.	More hardware than first two cases.
Four-address instruction	This is the most flexible of the four types, since it can contain three data addresses and a fourth address for the next instruction's location.	Same as above.

A few other forms of *multiple-address* instructions are shown in Fig. 7-6. These forms require longer instruction words than the *single-address* type. Many powerful processors continue to employ this instruction form, where the word lengths permit. Incidentally, the term *operand* refers to the data word to be operated upon.

Table 7-2 outlines some of the advantages and disadvantages to multiple-address instructions. These are more adaptable to computers having longer word lengths. The INTERDATA 7/16 has both two-address 16-bit and 32-bit instructions in its repertoire.

7-5 PROBLEM ANALYSIS IN PROGRAMMING

In order to program a computer to solve a particular problem the analyst must become thoroughly familiar with the problem itself and define the expected solution. He must first understand the mode form, interval of required inputs and outputs, computations to be performed, and whether magnetic tape, disk files, or printouts are required and in what formats. There may be special real-time considerations involved in the solution. It is all a very big order. The programmer must then lay out the necessary computations, and determine how rapidly and to what accuracy these must be performed to meet the solution specifications. Some questions to be asked include: Is internal memory sufficiently large? Can external files handle problem overflows where internal memory is overtaxed? Is the CPU fast enough? Does it have the necessary repertoire and other software available? Also, can suitable software be borrowed and updated where it is lacking? Where does all this effort fit into the software department's program preparation schedule? Are suitable *compilers* available to work with available peripherals? These and simular questions relate to the analysis and planning process.

It is possible to plan the program steps themselves once the preliminary analysis is completed. It may be best to divide the total job

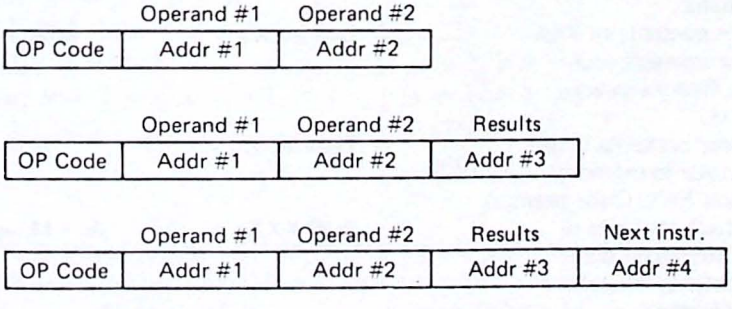


FIGURE 7-6. Some forms of multiple-address instructions.

among many programmers. There must be a carefully planned skeleton of a main program (or executive) in order for the separately prepared subroutines and subprograms to all fit (link) together. The executive and all subprograms are debugged separately with dummy data being used. At critical points in running each subprogram, etc. during debugging printouts are made to view the program statements. Such printouts are an invaluable aid for evaluating probable run results, and for ensuring that all components as well as the whole are operational.

Compilers normally will print out diagnostic statements when an error in program preparation is discovered. Error statements will print on the console teletype, etc. When the program runs without error statement printout, it has nearly reached its completely debugged status. Now the entire program (exeuctive and all the subroutines) are run together with ranges of dummy data provided to further check out correctness. After this, the program can be put to use or further modified. Documentation of the program now requires a final printout. If changes are made, a new printout is required. The program will now be filed on magnetic tape or a disk file, or, if it is short in length, will be put on paper tape or Hollerith cards.

TABLE 7-3 Table of instructions (partial list)

Operation	*Mnemonic*	*Programmer's shorthand*
1. *Transfer* from memory position XXX to accumulator.	TRA XXX	$M \rightarrow A$
2. *Multiply* contents of XXX with accumulator contents. Results in accumulator.	MPY XXX	$M \times A \rightarrow A$
3. *Add* contents of XXX to accumulator contents. Results in accumulator.	ADD XXX	$M + A \rightarrow A$
4. *Divide* contents of XXX by accumulator contents. Results in accumulator.	DIV XXX	$M/A \rightarrow A$
5. *Transfer* contents of accumulator to memory position XXX. Clear memory.	TRM XXX	$A \rightarrow M$
6. *Subtract* contents of XXX from accumulator contents. Results in accumulator.	SUB XXX	$A - M \rightarrow A$

7-6 PROGRAMMING STEPS

7-6.1 Sample Instructions

An instruction execution generates Boolean logic functions such as AND, OR, EXCLUSIVE OR, etc. in a timed sequence throughout the control gates of the processor. Each instruction has its own unique sequence. Table 7-3 lists a few sample instructions that one might find in a repertoire. These are in mnemonic form as would appear in an *assembler* listing. Practical computers have been built with as few as 12 instructions; however, these did more with each instruction, such as providing several addresses. Most modern processors, having single-address instructions, may have over 100 separate instructions. Note that the mnemonic form eliminates the necessity of writing instruction in (1) and (0) binary code—a tedious process and one prone to error.

7-6.2 Problem Solving and Programming

To illustrate the basics of programming, let us start with a simple example. Suppose that it is desired to compute the value of a simple algebraic equation

$$y = \frac{2x^2 + 3x + 2}{x + 11}$$

for values of x from 0 to 30. Te processor solves this step by step, inserting a value for x and computing the resulting y. For instance, the procedure is simply

Assume: $x = 0$	$x = 1$
Compute: $2x^2 = 2 \times 0 = 0$	$2x^2 = 2 \times 1 = 2$
Compute: $3x = 3 \times 0 = 0$	$3x = 3 \times 1 = 3$
Add: $2x^2 + 3x = a = 0$	$2x^2 + 3x = a = 5$
Add: $a + 2 = 2$	$a + 2 = 5 + 2 = 7$
Compute: $x + 1 = 0 + 1 = 1$	$x + 1 = 1 + 1 = 2$
Divide: $2/1 = 2$	$7/2 = 3.5$
	etc.

The process continues in this manner until the value y for $x = 30$ has been determined. This is all very simple.

The next important step is consideration of how to make the proc-

essor behave as desired. The answer, of course, is to write a program that accomplishes the desired task. This program contains the constants and designates places in memory to store the program, the variables, and the partial and final results of the calculations. Presumably one would like to view the results or to place the results somewhere else in storage—perhaps on a magnetic tape file. When one looks at the problem more closely, it is immediately apparent that the processor must accomplish several elemental tasks, such as:

1. Transfer numbers from memory to the arithmetic unit.
2. Add a number in memory on that stored in the arithmetic unit.
3. Multiply a number stored in memory to that in an arithmetic unit.
4. Divide a number stored in the arithmetic unit by a number stored in memory.
5. Transfer a number stored in the arithmetic unit to a specific memory location.

Being able to write such instructions in a form that the machine can eventually understand gives the programmer the capability to program his problem. Such a written instruction must have an *op code* and may have a *memory address code,* if one is needed. In addition, a successful program requires that each instruction be performed in its proper sequence.

To make the process easier to follow, let us assume mnemonic codes for all instructions as defined in Table 7-3. In Table 7-4 the steps of the solution are outlined, along with *op code* designations and memory cell assignments. On the right is a programmer's shorthand for defining the operations. It is assumed that the program is already loaded into memory. Let us look at the solution for $x = 1$. Table 7-3 indicates that 15 computation steps are required for $x = 1$ alone. For the 29 other steps for x computation, if each step took 15 instructions, the writing task would be rather arduous. Howeve, as we shall see later, there are ways to modify the value of x (the term is *increment*) and repeat the same set of instructions 29 times—each with the value of x incremented successively by 1. The addition of just a few more instructions to the present list makes this technique possible.

7-6.3 Symbolic Notation and Flow Charting

The problem of coding in machine language is one of having to memorize binary operation codes and addresses (1s) and (0s). An effective improvement is to convert to 0–7 octal code. Since octal code is close

TABLE 7-4 Program example

Step	Op code	Memory position	Coder's shorthand	Explanation
1.	TRA	100	M(x) → A	$x = 1$, in cell 100 moves to accumulator.
2.	MPY	100	M(x)•A → A	Multiply $x = 1$, in cell 100 by 1 in accumulator, x^2 now in accumulator. $x = 1$.
3.	MPY	101	M(2)•A → A	The quantity 2 in cell 101 multiplies $x^2 = 1$. Accumulator now contains $2x^2 + 2$.
4.	TRA	102	A(x^2) → M	The quantity $2x^2$ is moved to cell 102. Accumulator is cleared.
5.	TRA	100	M(x) → A	Same as step 1.
6.	MPY	103	M(3) →A	Multiply $x \times 1$ by 2 and leave in accumulator.
7.	ADD	102	M($2x^2$) + A → A	$2x^2 + 3x = 5$ is now in accumulator.
8.	ADD	104	M(2) + A → A	The quantity 2 in cell 104 is added to 5 in accumulator. It now contains $2x^2 + 3x + 2$, where $x = 1$.
9.	TRM	105	A(Y) → M	$2x^2 + 3x + 2 = Y$ is transfered to cell 105 in memory.
10.	TRA	100	M(x) → A	Same as Step 1.
11.	ADD	106	M(1) + A → A	The quantity 1 in cell 106 is added to $x = 1$ x now is 2.
12.	TRM	107	A → M(x + 1)	Cell 107 now contains $x = 2$.
13.	TRA	105	M(Y) → A	Y is returned to the accumulator.
14.	DIV	107	A(Y)/M(X + 1)	Y is divided by x + 1 = 2 in cell 107.
15.	TRM etc.	108	A(Y/x + 1)M	$Y/x + 1$ is returned to cell 108. The value stored is 7/2 = 3.5.

to decimal, it is much easier to memorize. The conversion from octal to binary is easily accomplished by inspection, as suggested in Sec. 2-3. For example,

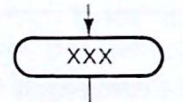
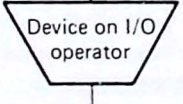
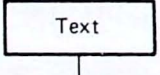
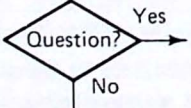
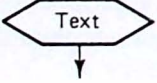
Octal notation		*Binary notation*	
Op code	*Address*	*Op code*	*Address*
54	*3412*	*101100*	*011100001010*

A further advance in symbolic coding is to assign mnemonic names to *op codes* that tend to define their action, for instance, ADD for addition, SUB for subtraction, MPY for multiply, TRA for transfer from memory to the accumulator, and TRM for transfer to memory. Even addresses can be given mnemonics to ease memorizing. For example, HOSS, HAND, NOTE, and CLIP could be typical memory position mnemonics. Special programs (*assemblers*) are employed to translate

from these mnemonics to actual machine binary codes recognizable by the machine logic. Mnemonic coding is used for further examples in this chapter. Also, memory cell addresses are sometimes given by their contents, as M_{xxx}, where XXX is the address. This notation can also be upgraded to M_{xxx} (data), where *data* is the contents of cell M. This is perhaps a convenient shorthand notation.

In the planning of a program a pictorial or graphic view helps one to visualize and keep track of necessary steps for the programming process. Flow charts perform this function. Normally two levels of flow charting exist. The first level plans the overall program, while the second-level charts define the program detail for individual blocks of the higher-level diagram. A higher-level chart appears in Fig. 7-7. At a later time the details of the blocks here may be detailed in second-level charts—even to the extent, possibly, of showing each instruction involved. First-level charts may show a complete subprogram or subroutine as a single block. It depends both on the application and on a programmer's preference as to how much detail appears on the charts. Table 7-5 summarizes a few of the more common flow charting symbolisms. There is a great deal of variation between users on exact symbolism; hence, the ones given here are, it is to be hoped, typical. Two additional examples of flow charts appear in Figs. 7-8 and 7-9. These respectively represent a first-level chart of a test sequence, and a second-level chart of a subroutine. Ideally, a flow chart is in sufficient detail that the appropriate *compiler* or *assembler* statements required are directly identified from the chart, if a second-level chart is assumed.

Table 7-5

Symbol	*Meaning*
	To START or STOP or to enter and exit a program at designated points. XXX will be START, STOP, HALT, or entry or exit symbols.
	This symbol designates an operation involving input/output.
	This symbol identifies an individual operation or step. *Text* can be any identifying information or symbols. If XXX is included, this block represents a complete subprogram.
	This symbol identifies a *branch* or decision point in a program. The program will follow the YES or NO path, depending on the results of the query—i.e., answer to the question posed.
	An operation that modifies an ADDRESS is represented by this symbol.

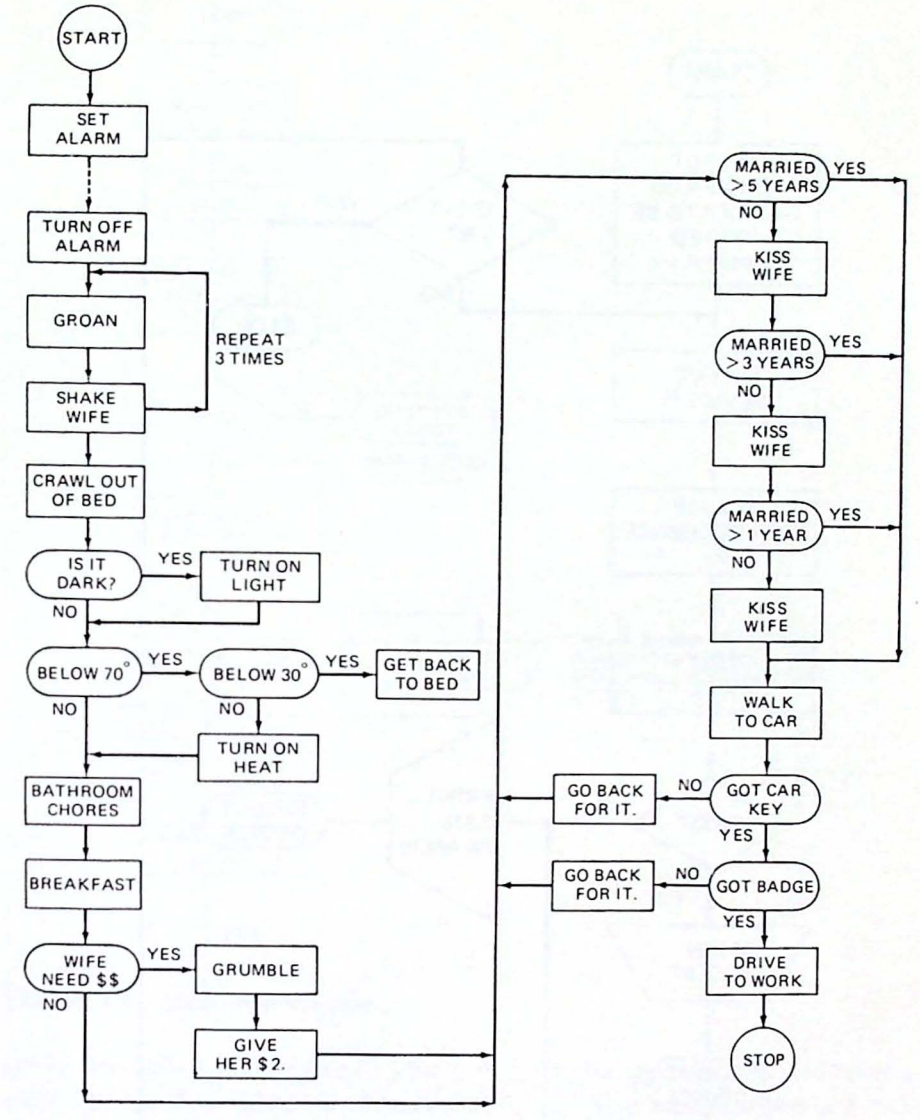


FIGURE 7-7. Flow chart—how to get to work.

7-7 DECISION BLOCKS AND THEIR USE TO CONTROL PROGRAMS

If one were to compute the value of $f(x)$ for each unit value of x from 1 to 1000, it would require a rather long program, if it is assumed that each step had to be identified by a separate instruction. Decision blocks avoid the necessity outlined above. In Table 7-5 the *question* block performs a *jump*, depending on the answer to a query. It makes it en-

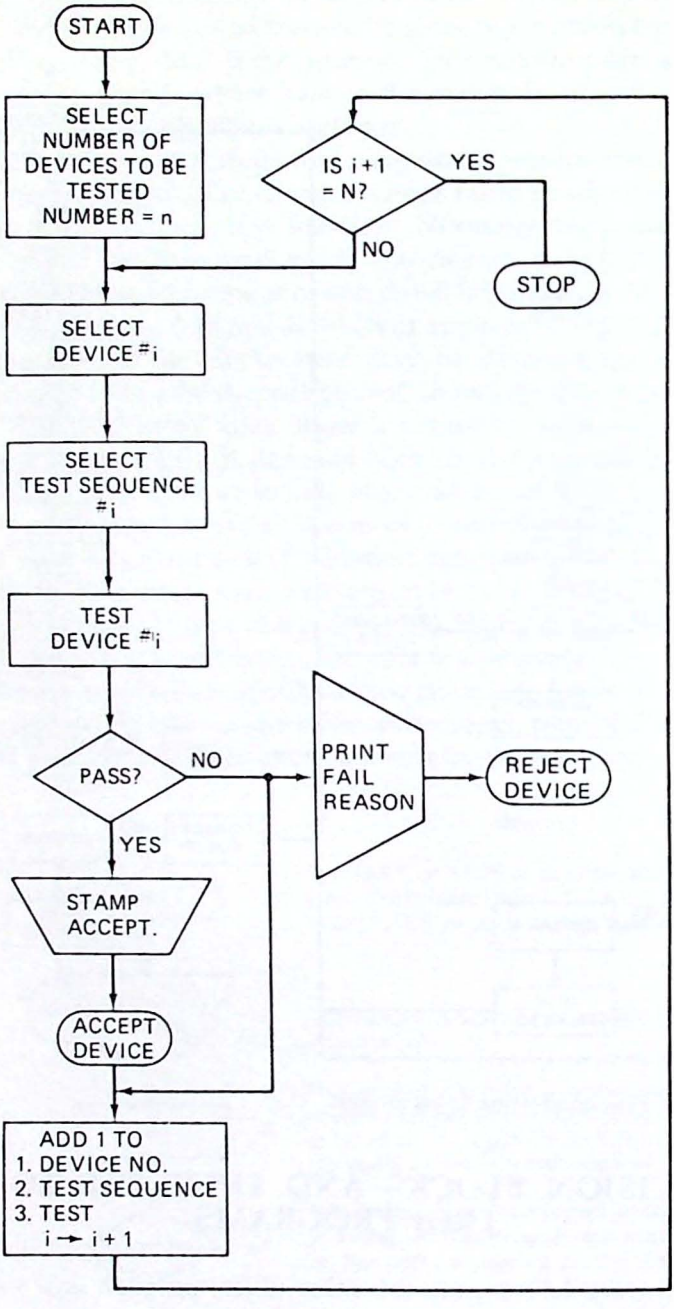


FIGURE 7-8. Test sequence of flow chart.

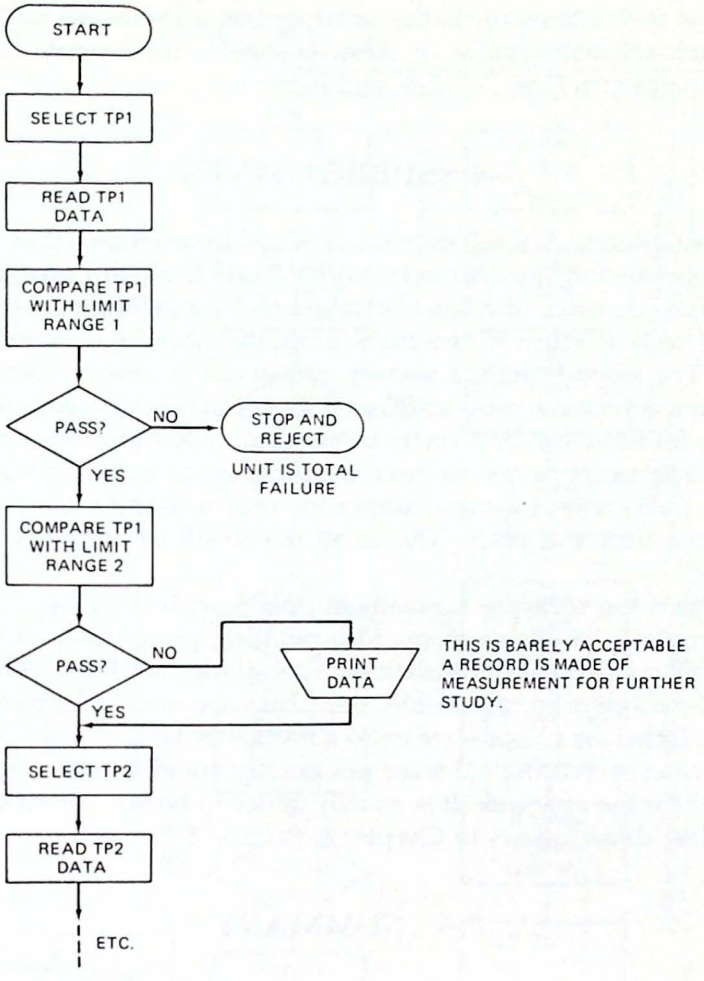


FIGURE 7-9. Subroutine flow chart.

tirely possible to compute $f(x)$ for $x = 1$, to change to $x = 2$ and recompute $f(x)$ for this value, to change x to 3, etc. This modification of x continues until $x = 1000$. Up to this point a comparison of x is made each calculation time to a stored value of 1000 by a second decision block. When the incremented value of x equals 1000, the second decision block causes the program to jump out of the computing loop to an address where the program will continue on. If this is the completion of the program, the jump will be to a location storing a HALT command. In the case where the program did not change the x value (did not loop), it would require several thousand instructions to do the job. With the jump-back comparison and incrementing method, it takes a compara-

tively few instructions to do the routine, but a thousand addresses to store each calculated value of data, if that is the intent. Jump-back routines appear in Figs. 7-7, 7-8, and 7-9.

7-8 SUBROUTINES

A subroutine is a self-contained set of instructions that as a unit performs some function. It can be injected into the main program every time it is to be used, or it may be called in by a calling statement, and released with a return to the main program after its function is completed. The second method usually conserves memory storage. Since many functions have been employed previously, it is likely that subroutines for these can be directly inserted or called into new programs. *Linking* statements permit an *executive* program to call or release a subroutine. Data computed via a subroutine return to the main program at the proper time and place. This is all the result of adequate program planning.

Within the software normally supplied with the processor, there are subroutines or subprograms for operating peripherals and providing special mathematical calculations, special file-handling routines, etc. These should be used, if possible, since they are usually proven and debugged. If there is no software to do a particular task, it must be written fresh, but even existing software not exactly suitable could possibly be modified for the new task. It is usually better to borrow or steal than invent. More detail follows in Chapter 8, Section 8-7.

7-9 SUMMARY

This chapter starts out by discussing classification of instructions according to their general type and organization of their field. The discussion centers on the Honeywell DDP516/H316 descriptions of *memory reference, shift, generic,* and *input/output* types. Other classifications could just as well have been employed, and often are by other manufacturers. Because so many of the larger machines employ multiple-address instructions, brief mention of these is included. The analysis process for planning a program is an extensive one, but can only be treated briefly here. The chapter continues with illustrative sections on programming (7-6.2 and 7-6.3) followed by a section on decision blocks—methods to shorten programs by looping—and concludes with a short explanation of subroutines and how they are included in programs. There is one additional thought that should be mentioned, and this is how variant bits are decoded for memory reference instructions. This action, although described in Sec. 7-2.1, is illustrated by the flow chart of Fig. 7-10.

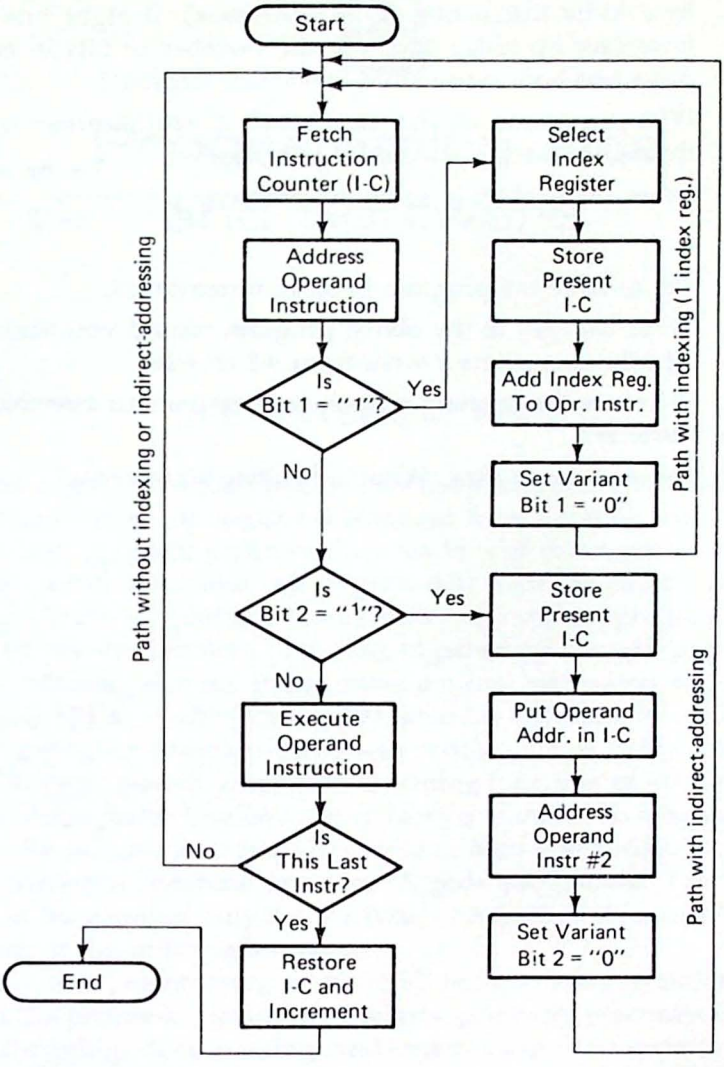


FIGURE 7-10. Variant-bit code processing. This figure shows the process of recognizing if a memory reference instruction is to be indexed, indirect addressed, or executed directly. If bit 1 is 1, the instruction is *indexed.* If bit 2 is 1, the instruction is *indirect addressed.* If both bits 1 and 2 are 0, the instruction is executed directly.

7-10 PROBLEMS

1. What are a *source program* and an *object program*?
2. Describe a shift instruction format: What bits are *op-code*? What bits describe the extent of the shift?
3. Describe types of I/O instructions. Define their fields.

4. In a 48-bit instruction (four addresses), if eight bits are allowed for op code, what are the number of bits in each address and how many words will they address?
5. What are some of the steps that a programmer must go through to write a successful program?
6. Draw a flow chart to solve the problem: $y = \dfrac{x^2 + 6x + 3}{x - 2}$
7. Try to write the program in H316 mnemonics.
8. What changes to the above program would you make to list all solutions where x went from +1 to +10?
9. What are the primary differences between an *assembler* and a *compliler?*
10. Define a subroutine. What are *linking* statements?

8

Programming Aids and Software

8-1 INTRODUCTION

The assembler or compiler allows the programmer to write his program in assembler (or compiler) language from a coding sheet. The assembler (or compiler) program then reacts with this *source program* and translates it into a *machine code program* that is loaded into the processor for actual problem solution. The *assembler* (or *compiler*) program handles automatically the problem of linking together the *executive program* and various subroutines, etc. If the *source program* statements are properly composed, the *assembler* (or *compiler*) eases the operation by constructing all the necessary program steps—sometimes a real help, since some steps can be forgotten when one is writing long lists of instructions in machine code form. Use of *compiler* (and/or *assembler* to a lesser extent) relieves the programmer from having to acquire a detailed knowledge of how a particular machine operates. A good programmer can function having at his disposal only the machine's programmer's manual and descriptions of the utility subroutines.

The term *programming* refers to all items relating to the programming of the processor, including analysis, planning, preparation, operation, debugging, documenting, and maintaining all programs. *Software* refers to the collection of programs available to the processor in solving both the specific problems assigned the processor and those service routines which ease the operation, the programming, and the maintenance of the processor regardless of its end use in a system.

It is the teaming of hardware and software that makes a processor so powerful in a system application. One without the other is of little use. Just as the design of hardware has advanced over the years, so has the design of programs. Because more emphasis has been placed generally on hardware design, it has matured more quickly; however, this emphasis is now changing. It is not unusual now for the software programming costs to outstrip hardware costs by a large margin. This is a complex subject and will not be discussed further at this time. However,

a few key terms will be defined and some general observations will be made to put *programming* and *software* in their proper perspective. However, the detailed explanation of program preparation was covered in Chapter 7.

First of all, the term *machine language* refers to a coding directly interpreted by the processor logic. Because this is in the form of (1s) and (0s) it is difficult for a person to learn to write code in this form. For the system to operate, however, the task programs must be in machine language. A program in machine language is called an *object program*.

Usually, the programmer writes his program normally in a near-English form. This program is called a *source program*. He employs the code form of an *assembler* or *compiler language* for which the machine has a *translating program*. The source program written in assembler (or compiler) language form is usually punched on IBM cards and entered, along with the translating program, into the processor. The translation routine then proceeds to derive an *object program* (in machine language). The object program is usually stored on magnetic tape and reentered as an input when the processor requires it to control the end process task. The steps described, of course, have been oversimplified to emphasize the important points. In graphical form the process steps are pictorially shown by Fig. 8-1.

Once prepared, the program (or segments called *subroutines*) is eligible for the software library. Here subroutines and/or complete programs are stored for further use or for inclusion into the other programs as they, in turn, are developed. Hardware manufacturers have realized the sales value of software libraries, and as a result, most of them offer extensive libraries along with their hardware.

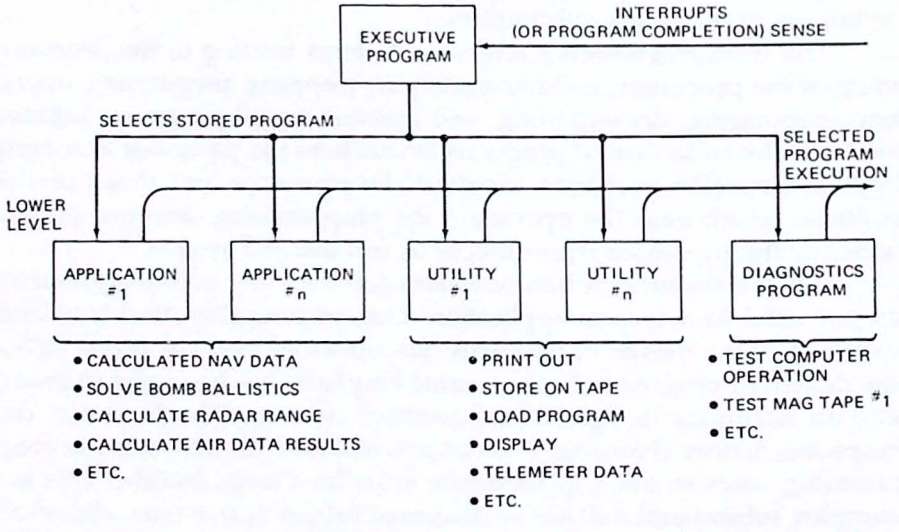


FIGURE 8-1. The programming process.

8-2 TYPES OF PROGRAMS

The usual programs prepared for a processor may be classified in the following categories:

- Operational (or application) programs
- Executive programs
- Computers and assemblers
- Utility programs
- Simulators

There are other classes that will be discussed in other sections, but this is a sufficient list for discussion at this time. The first two classes are generally written for a particular operation. They are, in a sense, special software to permit the processor to perform selected tasks. The next two types are more general and can be written by the manufacturer of the processor hardware to service all the processors of a particular type. *Simulators* may belong to either group, depending on specific circumstances.

8-2.1 Operational Programs

These programs are the ones usually prepared under the direction of the final equipment user. They may be originally written in *compiler* or *assembler* language; therefore, a translation to machine code is necessary before the object program physically performs the selected operation via the process hardware.

If a well-documented software library exists, it will aid in the preparation of the *operational program*. In this library should reside subprograms (subroutines or utility programs) that can be adopted into the main program flow of a new program. These subroutines typically include such suboperations as computing the sine, cosine, or tangent of an angle, code conversions from one code to another, and a set of instructions for operating the printer, for example. Well-defined operations such as these can be documented (with the proper linking cards) into the final program deck. To recap, then, an operational program is a collection of individual instructions—data and subroutines put together as a package to perform (when placed in the computer) a defined task.

8-2.2 Executive Programs

The *executive program* is a higher level program than the individual *operational program* just considered. Figure 8-2 illustrates its primary function of supervising a group of operational programs and *utility pro-*

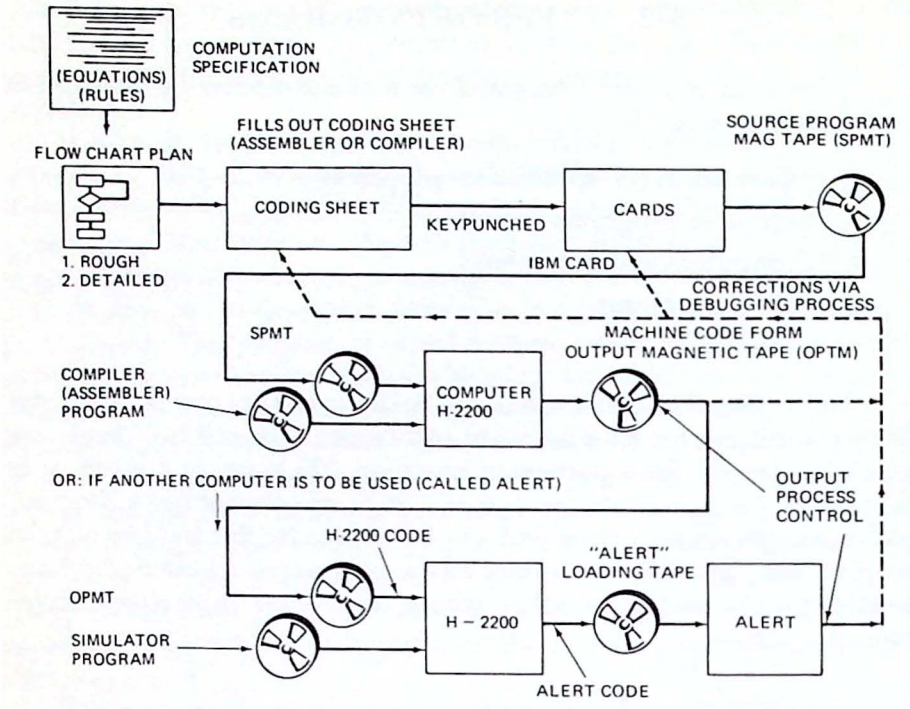


FIGURE 8-2. EXECUTIVE program features.

grams in a situation where more than a single operational program is involved. Many processors operate in a multiprogramming mode; i.e., more than one program is located in memory during the processing operation. The executive program, then, has the task of selecting which operational or utility program will operate, and when, and for how long. This selection is made on the basis of priorities previously established. *Interrupts* from a peripheral may request that the processor transfer data. The executive program then switches to a utility subroutine that services the peripheral, but in such a way that the presently operating program is delayed as little as possible in its continual execution. There are cases, of course, where the executive senses a condition where an entirely new operational program is required, and, therefore, will put this new program in operation instead of the present one.

8-2.3 Utility Programs

Utility programs are the programs furnished by the manufacturer usually to perform a number of relatively routine tasks. They are program routines that check out the computer, perform standard computation functions, help with the diagnostic and debugging (if any problems

are discovered during installation), and handle such operations as initial memory loadings, peripheral controls, and error sensing. If a special operational function is performed by a program complete in itself, the program is often termed a utility program rather than a subroutine. It is a matter of judgment, however, in many instances, whether a program is classified as a subroutine or a utility program.

For instance, a routine that allows the system to merge several files and rearrange them in alphabetical order could be termed a utility program because of the complexity of the final program. In general, a subroutine will be a relatively small, unique set of instructions that form an integral portion of the larger program.

8-2.4 Simulators

The term *simulator program* is not as easy to define as it appears. In this section we will define a simulator program as one that permits one to simulate operation of one computer on another. Occasionally it is used in a much broader context—meaning the representation of any system element via a computer program.

One of the major uses of simulator programs is in the design of new computers. The instruction repertoire of the new proposed machine can be approximated by simulating the new instructions on the existing machine. In this way, the timing and functioning of the new machine, when solving problems in its intended application, can be studied and evaluated before hardware is committed.

Another use of simulator programs is to prepare programs on one computer to run on another computer.

Many small processors are not large enough to have their own *compilers*. As a result, they must be programmed presently in *assembly* or *machine language*. It is often much easier for the programmer if he can write a program in compiler language on a machine that has a compiler and then by the simulator program have the first machine punch a tape (now in machine language) to operate directly on the second machine having no compiler. This general approach is much used for military mobile processors, which have relatively small memories when compared with other control processors.

8-2.5 Assemblers and Compilers

Assemblers and compilers are the family of programs that translates a programmer's coding sheet statements into a language (machine code) that the processor can interpret. The coding sheet statements are in an alphanumeric mnemonic form (a *source program*) written in a spe-

cific code appropriate to the assembler or compiler involved with the translation. The assembler (or compiler) program is first loaded into the processor memory, and this becomes the translation generator. The source program, e.g., is converted by a keypunch operator to IBM cards. It is still in the alphanumeric format of the source program at this point.

The compiler or assembler process is then performed, and the generated machine code is stored in a vacant section of the memory. This machine code program (now the *object program*) can then be collected on an output tape. This tape is then reinserted at the processor input as the *operational program* for the task to be accomplished. This entire process is illustrated by Fig. 8-3.

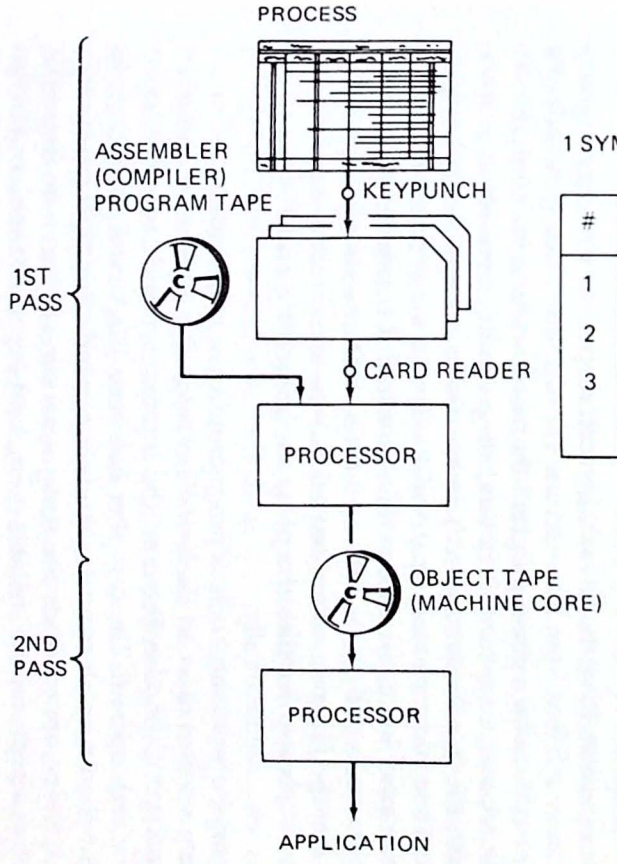
The general differences between an assembler and a compiler are important. An *assembler* is usually composed solely for a particular processor or a related line of processors. It converts one symbolic statement of the *source program* into one statement of the *object program*. The basic objective of an assembler is to develop a means to write programs in a symbolic code that is easy for the programmer to understand and recognize. It is easier to write and remember an instruction such as DOG-PLACE, where DOG represents an operation and PLACE a memory address, than 10111010011. Ones and zeros look pretty much alike, and it is hard for most people to clearly rmember the codes required, let alone to write them correctly every time. Why not let the machine do the translating? This is exactly what an assembler does.

The *compiler* is a level again more powerful than an assembler. Whereas the assembler generates machine code statements on a one-to-one basis, the compiler, on the other hand, may generate several machine statements from a single input symbolic statement. This means that a programmer under the proper conditions can generate a relatively long machine program sequence with just a few symbolic statements. Compilers were first designed to solve arithmetic-type problems. The programmer can direct the process to perform a rather complex evaluation, for instance:

$$y = \frac{(ax + B)^2 cx^2}{A^2 + B^2}$$

by writing a single statement. The statement must be written in the correct compiler form, however, which differs somewhat from the above. Also, he must designate values for A, B, C, and x. Assemblers and compilers are compared in both Fig. 8-3 and Table 8-1.

Compilers are written for categories of applications rather than for a particular machine. For instance, FORTRAN, a popular compiler, is employed on a vast number of machines of different manufacture. Another advantage of compilers is that, if a person learns it on one



CHARACTERISTICS

ASSEMBLER

1 SYMBOLIC INSTR. = 1 MACHINE INST

	CARDS	OBJ TAPE
#	SYMBOLIC FORM	MACHINE FORM
1	ADD, DOG	1101 1100
2	SUB, MOM	1001 1101
3	PUT, FOX	1010 0001
	ETC.	ETC.

COMPILER

1 SYMBOLIC INSTR = SEVERAL MACHINE INSTR'S

#	SYMBOLIC FORM	PROGRAM STEPS
1	LRC = (X + Y)	
	••2• Z	START
2	X = 1	$X \rightarrow M_1$
3	Y = 10	$Y \rightarrow M_2$
4	Z = 5	$Z \rightarrow M_3$
5	WRITE LRC	
6	EXECUTE	$M_1 \rightarrow A$
7	STOP	$M_2 + A \quad A \rightarrow = K$
		$K \rightarrow M_4$
		$M_4 \times A \quad A \rightarrow = K^2$
		$M_3 \times A \rightarrow A \quad K^2 Z$
		$A \rightarrow M_5$
		SELECT P
		MOVE M_5 P
		STOP

FIGURE 8-3. A comparison of an assembler and compiler program.

TABLE 8-1. A comparison of assemblers and compilers

Definition		
The *assembler* is machine-oriented and converts *symbolic* programs to *machine* programs (one to-one basis).		
The *compiler* is operation-oriented and converts *symbolic* programs to *machine* programs (one-to-several instruction basis).		
Operational Use		
The user writes *application* program in assembler/compiler language. The assembler/compiler converts application program to machine language.		
Examples		
Assemblers	*Compilers*	
SNAP	FORTRAN IV	
DAP-16	COBOL	
EASY-CODER	ALGOL	
	JOVIAL	
Advantages		
Easy to learn		
Automatic control of operations		

machine, the steps will differ little for another machine. Even though there probably will be some minor differences between the same basic compiler used on machines of different families, these differences are easily learned.

The *assemblers* have the advantage with respect to writing programs that are more efficient from the machine standpoint. They generally use fewer instructions for a given program than one written via compiler. To state this another way, compilers tend to generate more *object program* steps than the minimum required; hence these compiler-written programs will take more memory space and will take longer to run. *Compilers,* on the other hand, are more efficient when it comes to writing the *source programs* initially, since fewer written statements are usually required, and the program so written takes into account the operation of the peripherals and handles the program branching and formatting of readouts, etc., automatically.

In some operations, such as programming at a time-share terminal, a program referred to as an *interpreter* occasionally replaces a compiler. The important distinction between the interpreter and compiler lies in how they each operate. The compiler has been discussed; therefore, its operation should be clear in the student's mind. For this interpreter, however, the computer reads the statements directly from an input device such as a paper tape, translates them, and executes them, but in this

case the computer executes each statement completely as it is read before going on to the next statement. Thus the process of interpreting is usually slower than compiling.

In this book we will not discuss interpreters to any extent. The term was introduced to emphasize that special program-oriented languages exist and are in wide use, but are not of the class of a highly versatile compiler.

8-2.6 Generator Programs

This is a program that is job-oriented. A *generator* produces a specific program from the specifications of the problem. It is different from a compiler in that it is directed at a specific task. Thus, a program that generates a published report, given the specifications of the report, i.e., the outline, subtitles, titles, tables, etc., is a *generator program* and in this specific case, a report generator.

8-3 DEBUGGING PROGRAMS

Hardly is one lucky enough to have a program, if it is at all extensive, to work the first time. There is normally some debugging (getting out the goofs or bugs) to be done. Actually, somewhere between one-quarter to one-half the programmer's time is normally spent debugging a so-called finished program. Part of this is due to the fact that an active program is never quite finished—there are always changes requested, and these can lead to unexpected, but, it is to be hoped, essentially minor problems.

The debugging process involves running the program on the processor and observing the results. Simulated data, or data that will give known results, are employed to test the computational algorithms and other aspects of the program. When unexpected results occur, the programmer attempts to diagnose the reason for these discrepancies in order to understand what is wrong. Optimally, he will quickly find a clue that points to the necessary program correction. The process is continued until the entire program is free of bugs and sample runs give plausible results with simulated data.

Several software aids are generally available to the programmer to help him in his debugging task. First of all, he may request a printout of the assembled program along with the machine codes. Thus, he can determine what exactly is stored in memory initially. Second, he can design the program to frequently print out partial results for analysis. Third, he can call on software programs to print out either the entire memory contents or preferably, a smaller portion directly concerned

with his program. These programs are frequently called *dump programs*. Also, other programs, called *tracers*, will print out the contents of the important registers periodically on demands of the programmer or when the program fails to run. All or one of these aids can be very helpful. In addition, the processor usually has provision for inserting single commands by the operator, one at a time, which allows him to test one step at a time for specific conditions. If he suspects that a particular command fails, it is very helpful to try out this one command alone and observe the immediate results of its faulty execution.

Debugging is often accomplished in stages. The first milestone is when the program runs completely the first time. The second important event is when it runs completely with normal data. The third is when it runs with every conceivable combinaton of data it is ever expected to be confronted with. Fourth, the last unanswered questions are: Will the program run in conjunction with or as a part of the total system program? Will it mesh properly as part of the system's package?

Although it would be most convenient for the programmer to debug his program online at the computer console, quite often he is not permitted to do so. It is just too expensive to allot the computer for this purpose—especially when other programs are in the process of running. Therefore, he probably will submit his program card deck, etc., to an operator, who will sandwich it into the schedule at a convenient time. The operator will run the program and then report back the results: how it behaved, what went wrong, etc. With this information, suitable modifications will be made and the processes repeated until success is eventually achieved.

8-4 COMPILERS AND ASSEMBLERS

8-4.1 General

A *compiler* program is one which converts a program written in a carefully prescribed *source* language to a *machine language* program set of instruction codes. This latter program, when entered into the computer, performs the desired programming task. An *assembler* program also performs this function, but there is an important difference between a compiler and an assembler. The *source language* program, in theory at least, can be compiled on any computer having among its software the particular compiler program. For example, FORTRAN, COBOL, PL/1, etc., are available compilers for literally hundreds of computers. Programs written in these, and other compilers too, may be compiled and run on a number of machines. Theoretically, programs in compiler language form are independent of a particular machine, although in practice this is not quite true, because compilers tend to differ slightly between software of

different capabilities and are designed for a different classification of tasks.

Basically, a compiler takes a single source statement and generates a batch of machine language statements.

8-4.2 Assembler

An assembler is more limited, i.e., one machine language statement generally is derived from a single *source language assembler* statement. Thus, the assembled machine program is likely to be very close in form to the *assembler language source program*—more so, probably, than a machine language program generated by a compiler. Assemblers are oriented to a single computer, or at the most, to a single computer family, each one of the family using the same instruction set. For example, the Honeywell DDP116, DDP516,H316, and HDC601P all use the DAP16 Assembler, orginally derived for the DDP116 computer back in the early 1960s. The other computers mentioned do in fact employ the same instruction set; thus, programs assembled for the DDP116 run substantially unaltered on the DDP516, H316, and HDC601P computers. A new computer in the series, the H716, will funcion on programs for the older computers; but it, in addition, has some additional instructions. Thus, programs written using the new instructions will not run on an H316, etc.

Generally, an assembler assigns a memory location to each constant and variable in the program. It also generates the necessary instruction list, assigning the proper machine codes and addresses to each generated instruction, and provides for the necessary address locations or relocations. The process is primarily one of table look-up operations. Macro instructions may be included within the assembler to generate specific formats on the machine code tape, but these do not run as part of the operational program itself. The assembler for the H316 is furnished either as a one-pass or two-pass assembler. The two-pass assembler, although not as quick, does generate a more efficient code with fewer wasted instructions.

As pointed out previously, assembers are programs that help the programmer write programs. This sounds redundant, but it is truly the function of an assembler. With the availability of an assembler program on a particular processor, it is possible to write the individual instructions and data to the processor in symbolic form; the *assembler* program converts these symbolic or mnemonic instructions, data, etc., to machine languages and assembles them as a machine program. Also, the assembler generally causes a printout of both the original set of symbolic instructions and the final machine codes. This enables the experienced programmer to inspect both sets and check for possible errors. In

addition, assemblers can be written to run on processors other than the one to be finally programmed. Thus machine language *input tapes* may be obtainable directly for a machine that does not have an assembler of its own. As an example, the input program tape for a military control processor can be prepared and checked for correctness on a ground-based central processor. For this reason, it may never be necessary to write a special assembler program operable on the military processor itself—although it may be useful to do so at a later date.

Because the concept of assembler is a complex one and a subject worthy of a book itself, only a single example will be presented here. This example is summarized by Fig. 8-2.

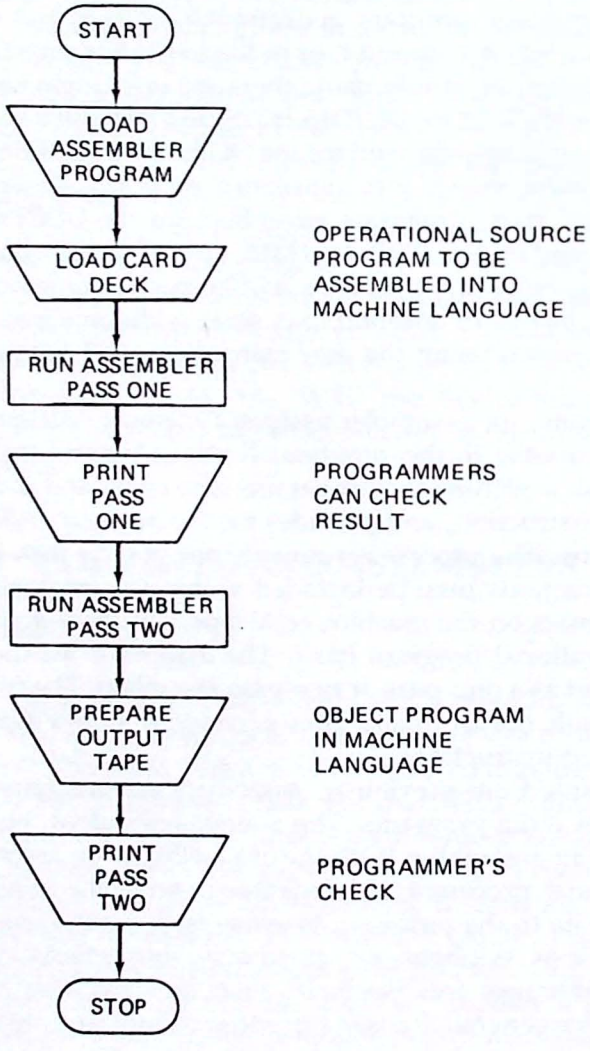


FIGURE 8-4. Assembler program example.

By way of explanation, the interpreting assembler program is assumed to be already written and stored previously in the processor memory. Also, the new user program (in a symbolic form that the assembler can interpret) is available as a deck of punched cards. At this point, the intention is to convert the source program (the new user program) to a machine language program that actually performs the application process.

Normally, the assembler operates in two passes. The deck of cards is read sequentially into the memory with the assembler specifying the starting point in memory and putting each successive entry in ascending memory cells. The assembler then performs its *pass one* task of assigning a memory cell address to each program constant value (LARK, LIGHT, LEMP, etc.). Whenever the source program refers to one of these constants again by its symbolic name, the program refers it to the assigned address of this constant.

The card deck is again read into the machine. On this second pass (pass two), the symbolic operation code of Step D (see Table 8-2) is compared with a stored look-up table, and the machine operation code is assigned in place of the symbol. The machine operation codes are then assigned memory cells in program sequence, and the addresses of the constants are retrieved and assembled together with the operation codes. The result is a sequential set of machine instructions complete with operation codes and addresses, as shown in Step C of Table 8-2. This assembler program now causes a printout of both the original source program and the assembled machine program. At the same time, a corresponding magnetic tape (or other storage form) is normally prepared by the processor as an output. This is now the *operational program* and will be reinserted again into the processor to run the assigned operation task.

TABLE 8-2 The Assembler Process

Program to be assembled

```
A LARK = 500
A LIGHT = 250
A LEMP = 0
  TRA LARK
  SUB LIGHT
  DIV LEMP
  PRI A
```

Store source program in memory

The program to be assembled is read into memory by a loading program. The *assembler* program (which actually does the assembly)

(Cont.)

TABLE 8-2 (Cont.)

is also read into the memory. Thus the first step is to store the above program:

*Memory Address**	*Store*	*Memory Address**	*Store*
50	A LARK	55	10
51	500	56	TRA LARK
53	A LIGHT	57	SUB LIGHT
53	250	58	DIV LEMP
54	A LEMP	59	PRI A
			.
			.
			.
			etc.

**Address in memory where program is initially stored.*

First Pass of the assembler

The assembler takes the *A* labeled items and stores them starting at an assembler known address in the form of a *look-up* table. An address is assigned each item. Future reference to the mnemonic will be referred to the table, where the equivalent address will be substituted; hence,

*Memory Address**	Variation	Assigned Address
1000	LARK	150
1001	LIGHT	151
1002	LEMP	152, etc.

The second pass involves assembly of the program

The values of the variables are inserted into the assigned addresses as follows:

*Memory address**	Quantity
150	500
151	250
152	10, etc.

Also, a table look-up process is used to replace the op code mnemonics with binary (here expressed in octal) codes. For example, TRA = 10, SUB = 23, DIV = 14, PRI = 11, etc. Therefore,

	IC	*Op code*	*Address*		*IC*	*Op code*	*Address*	
	56	TRA	LARK		556	10	150	
Source	57	SUB	LIGHT		557	23	151	
Program	58	DIV	LEMP	becomes	558	14	152	when
	59	PRI	A		559	11	—	assembled

**Memory address is via instruction counter (IC).*

8-4.3 Macroinstructions

The use of an assembly language symbolic coding does not decrease the number of decoding steps, since there is one symbolic instruction written for each machine instruction to be generated. In many instances, the same group of instructions is employed in the same sequence time and time again. For example, suppose that it is required to input from a character-oriented peripheral device, but the characters are to be packed two at a time in a single computer word. This same process is used frequently. The action as described could require several program steps, but they are always the same ones.

In some symbolic systems, the individual program steps need not be written into the program each time they are required. Instead, a new instruction is defined, one that may not appear in the original repertoire. This new instruction calls for our sequence of individual instructions. When this sequence is to be used, the programmer writes the new instruction by itself. This new instruction is termed a *macroinstruction*. It is the basic ingredient of many compilers. Although a macroinstruction plants lists of instructions in the program, every time it is employed (like a closed subroutine), the added length to the total program is not usually crucial. The alternate technique of jumping to a subroutine can save total space, since only the linking instructions are added to the main program each time the subroutine is to be used.

8-4.4 Assemblers and Macroinstructions

The typical assembler permits a programmer to write the individual steps of the *source program* using the assigned mnemonics for operation code and the addresses in symbolic form. This particular phase of the process has been illustrated previously in Sec. 7-1. If the *assembler program* were restricted to this alone, it would ease the programmer's task some but not very much. However, by using *macroinstructions*, the assembler assumes new powers one might not expect of it. Special control instructions are added which assist greatly in organizing the final *object program*. The instructions, sometimes termed *pseudo-operations* rather than *macroinstructions*, are written into a program in the same way that the normal machine instructions are written in symbolic form and follow whatever rules of procedure are required. As an example, pseudo-operations included in one widely used assembler (DAP-16) provide the following control features:

1. To start, stop, and select the assembly modes. Included among the pseudo-operations are the setting of the program counter

(starting point of the program in memory), whether the addressing is to be absolute or relocatable, and other program control functions.

2. To define the form of data, and instruct the program on how the data are to be handled, i.e., binary, octal, decimal, floating-point, etc.
3. To control the loading sequencing from the peripherals. This function would include the assignment of memory, the establishment of base sectors in a sectionalized memory organization.
4. To provide a callup of subroutines and to provide an exit back to the main program.
5. To specify the listings provided by the printer and similar formatting on other peripherals.

These five examples are typical of the added convenience that a well-designed assembler provides the programmer. In addition to the *object program* in paper tape, IBM card, or magnetic tape form, the assembler will provide a complete listing printout of the program step by step as it occurs in final assembled format. This list will include the original symbolic source statements and comments; if any, the operation code and address in machine language format; and the position of the program counter, most likely the latter two in octal codes. If the program fails to assemble properly or if there are errors, error messages will also be printed if the program detects these errors. Obviously, such a complete listing is indispensable in checking to see what might have gone wrong when a program contains errors. This listing is a most useful debugging tool.

8-5 COMPILERS

A compiler, on the other hand, will examine a *source instruction*, decide on the steps to take, and assemble a set of machine instructions to accomplish the task. For example, a source statement might state OUTPUT TO PRINTER. The machine's program will probably require a number of interrogation steps such as: Is the printer ON? Is the buffer ready to accept character codes? Which printer is it (#1, #2, or #3)? Is the line to print completely loaded in the buffer? Is the printer presently printing? The program will have to set the format and decide what to do when a page is filled. The OUTPUT TO PRINTER will be replaced by a JUMP instruction, which will cause a program jump to a particular subroutine whenever OUTPUT TO PRINTER appears in the *source program*.

After the PRINTOUT task is completed, a return to the main program is implemented. Thus, compilers self-generate programs that are basically sets of subroutines linked together. It is normally easier to write an application program in compiler language, but the machine code is likely to be more complex than one written in assembler language. Also, some desired operations may not be covered by the compiler language and modifications to the *compiler* must be made, or a return made to assembler language programming for these special operations. If the source program wished to compute

$$Z = \frac{[xy + 2ax\,(a + b^2)]}{4cm}$$

the operator would write a suitable FORTRAN statement.

The compiler generates a subroutine which would assign values to x, y, a, b, m, and c, compute and store xy, compute and store values for $2ax(a + b^2)$ and $4cm$, probably in a similar manner. The steps to perform xy include:

1. Transfer y value from memory to the A register.
2. Transfer x to M register.
3. Multiply A register and M register. Results are put in A register.
4. Transfer A register to memory cell XXX5.

For the term $4cm$:

1. Transfer value 4 from memory cell to A register.
2. Transfer c value to m register.
3. Multiply 4 in the A register by c.
4. Transfer m to M register.
5. Multiply contents of M register by A register and leave c results in A register. Transfer A register contents to memory XXX6.

For the $2ax(a + b^2)$ term:

1. Transfer b to A register.
2. Transfer b to M register.
3. Multiply M register by A register; leave results (b^2) in the A register.
4. Put a in the M register.

5. Add M register to A register. Leave results in A register.
6. Transfer A register via M register to memory cell XXX7.
7. Put X in A register.
8. Put 2 in M register.
9. Multiply M register by A register and leave $2x$ in A register.
10. Place a in M register.
11. Multiply M register by A register and leave results in A register.
12. Multiply M register (memory XXX2) by A register.
13. Transfer A register to cell XXX8 via M register.
14. This puts $2ax(a + b^2)$ in memory cell XXX8.
15. Put XXX5 contents to A register.
16. Put XXX8 contents to M register.
17. Add A and M registers, leaving XXX5 + XXX8 in A register.
18. Move XXX6 to M register.
19. Divide A register by M register.
20. Puts results in memory cell XXX9 via M register.

Thus, memory cell XXX9 contains $Z = \dfrac{[xy + 2ax\,(a + b^2)]}{4cm}$

Somewhere in memory the values of x, y, a, c, and m must be stored, because the compiler statement calculates a numerical value. Thus, a compiler such as FORTRAN is a powerful aid in writing programs, since a single FORTRAN statement generates a subroutine that is equivalent to approximately 29 assembler statements. No wonder the higher-level languages, where they exist, are most used by programmers.

$$Z = X \times Y + 2 \times a \times X(a + b \times b)/4 \times c \times m$$

Would be a suitable FORTRAN statement for the above problem. A brief description of FORTRAN is given in the next chapter.

8-6 SUBPROGRAMS

Subprograms are segments of programs that perform a defined task or operation and, if properly formulated, can be inserted as a unit in other programs where the same task is to be performed again. These subroutines can be stored and used over and over again. Subprograms

are classified according to their task. For instance, the square root calculation, integration, calculation of the hypotenuse of a right triangle, and generation of an angle's sine from a series could each be examples of an arithmetic-type subroutine. Examples of other types of subprograms could be turning on and off peripherals, accessing data from a peripheral, measuring test voltages automatically and numerous other operations of similar types that can be processed by a modular set of instructions.

In many cases, it is desirable to include a subprogram as part of a new program. What is needed, then, is to seize the complete subprogram and sandwich it in its entirety into the main program at the proper point.

An *assembler program* or *compiler program* if it is versatile, will have call and return statements that will permit a complete subprogram to be placed conveniently within a new program and will also allow a return to the main program, respectively, when this subprogram has finished its task.

There are two ways to link subprograms to the main program. The first method is called the *closed method*, whereby the subprogram physically becomes a part of the main program. In this method, every time the subprogram is to be reused, it is inserted directly into the program at the point of use. This method tends to lengthen programs and is only suitable in the program in which it is inserted.

A more general approach is to use *open* subprograms. These subprograms are stored separately in memory, and call statements are provided that permit a program jump to the subprogram whenever it is needed. Thus, any main program during execution can call the routine as needed, and its redundant storage is thereby eliminated. Once the subprogram has completed its task, a return jump is provided by a return statement back to the proper point of the main program. The main program then continues from this point as if no interruption had occurred, using the results computed by the subprogram.

8-7 SOFTWARE

Software involves the total programming package available to the users. Nowadays, the processor manufacturer himself will normally supply a basic package of programs that generally include such items as a *loading program*, and *assembler* (possibly a *compiler*), program *debugging routines*, *hardware debugging routines* (*diagnostics*), routines to operate selected peripherals, and subroutines to provide special arithmetic computations, control, or business operations. Among this list may be an *ex-*

ecutive program (or *supervisor*), *dump and trace programs, edit programs, arithmetic function programs,* and *common application programs.*

Unless the user's application is rather routine, he can expect to require new special programs relating to his application alone. This *operational program* is not usually provided by the manufacturer. If the user is lucky, perhaps an existing operational program can be modified to fit his case. However, it normally means that the user must hire programmers or engage a service organization to provide the custom operational programs he needs. This expense can be a substantial portion of his total investment costs, and it should not be ignored when the overall cost of the processing installation is estimated.

To assist each other, users have formed user groups whose function is to exchange experiences, operational data, subroutines, subprograms, and actual programs. Quite often, the manufacturer is the sponsor, since it is to his everlasting benefit to encourage wide use of his processor line in as many applications as possible. Records are kept of user programs and their availability. Newsletters are circulated to the members. Even user conventions are occasionally held. In this manner a wider distribution of experience and workable programs is available than would be possible without such a means of personal contact.

The collection of programs, subroutines, subprograms, documentation, etc., that is available to the user comprises his software library. Ideally, when a new program is needed, much of this collected material can be adopted as segments of the new program. Because punched cards or tapes stored in the library look very much alike, it is necessary to document each program and subroutine carefully in order that its organization, usage, data refinements, and limitations are recalled. The programmer who reuses a particular library item is likely not to be the one who wrote it originally, and even if he is, he will have forgotten the details. Since changes can be made only with reference to known conditions, complete documentation is necessary to define these reference conditions. Since program writing is still somewhat of an art and no two programmers write in an identical manner, only good descriptive information allows a complete understanding of the original work. For some subroutines or subprograms it may not be necessary to list all the program steps, but complete specifications of how it is to be used must be prepared. Programs of any appreciable length will be prepared in modules by separate programmers working by themselves generally. In the end, their work all must mesh properly. Consequently, descriptive material must be prepared to describe the work to others and thereby eventually to permit an overview of the total creation. Well-documented records (for example, a printout of the assembled program) aid both in merging of the modules and the later inevitable debugging process.

8-7.1 Manufacturer Supplied Software

Summarizing, then, manufacturer supplied software below the *executive program* level can typically include the following:

8-7.2 Loading Programs A most important form of program is the *loading program* or simply *loader*, as it is sometimes identified. Without these programs, one could not insert any programs into the machine itself. More than a single-type loader can be available among the software. Some typical types include:

Bootstrap. This form of loader usually consists of a few instructions inserted manually, one instruction at a time. Usually it is inserted by buttons and controls on the control panel. This program is designed to load into the memory other and more complex loaders that will accept programs residing on magnetic tapes, paper tapes, cards, etc. These more versatile loaders can be either absolute or relocatable.

Absolute loader. This loader, when it is in core, loads a program starting at a specified location in core memory. The starting point is always the same unless the program itself is modified specifically by the programmer.

Relocatable loader. This program loads another program into memory at a location that is not initially specified by the programmer. In fact, he may not know where it is to be loaded. The starting point location is determined by the *executive program* after it has examined the present location of programs destined to remain in core, at least for the moment, and the other space that is available. This operation appears to be automatic. It is often a necessary function in multiprogramming and other time-sharing operations, where complete files are continually moved in and out from disk and magnetic tapes during the processing.

Loaders are among the software normally available from manufacturers for the peripherals they supply.

8-7.3 Peripheral Utility Subprograms Peripheral utility subprograms control the operation of the selected peripherals. They start, stop, test for correct operation, check parity, and transfer data between memory and the peripheral devices.

8-7.4 Interrupt Servicing Subprograms These subprograms are instigated by the receipt of an interrupt signal. They determine if the inter-

rupt is the one having the highest priority and, if so, it jumps to the program specified. This process involves storing the present contents of the processor registers, the instruction counter, the accumulator, etc., of the interrupted program temporarily in memory during the processing of the interrupt subroutine. The interrupt subroutine now loads the interrupt program into the processor and proceeds with the operation. At the end of the interrupt program, the original program is restored.

8-7.5 Mathematical Subroutines These subprograms compute or store tables of special mathematical functions, which can vary all the way from double-precision mathematical operation to the computation of special and complex mathematical functions. Such functions as the $\sin x$, $\cos x$, hyperbolic tangent, tables of logarithms, and antilogs can be computed as terms of a series or stored in tabular form. Complex operations such as integration, differentiation, and inversion of matrices may be carried out via suitable subroutines either stored within or called to the program when needed.

8-7.6 File Management Subroutines These subprograms generate files and allow files to be searched and updated and data to be added or deleted or extended. These programs merge old files, create new ones, and delete partially or completely ones no longer of any use. They also relocate files and permit files to be distributed rather than concentrated in a specific order sequence.

8-7.7 Dump Subprograms These subprograms are the opposite to the loading programs discussed previously. They permit portions of the main memory to be outputted to magnetic tapes for storage or to a printer where the stored program can be observed directly as a printout.

8-7.8 Diagnostic Subroutines These subprograms are employed to test both hardware and software for errors or improper operation. They can be used to check out system operation initially and during system installation. They may be loaded when required to test for malfunctions or can be incorporated as a subroutine within the main program.

8-7.9 Software Packages As a final example, Table 8-3 lists a few of the typical software packages that a business user might expect to procure from the manufacturer of a business-processing system. The first two classifications are of general usefulness, and most users require them. The third class is more specialized, and a particular user may not be able to apply them without extensive modifications to either the package or to his operating system. For this general reason, application-dependent software tends to be less than satisfactory for all but a fraction of new applications.

TABLE 8-3. Typical Business Processing Software Packages

1. *Utility packages*
   - Input/output routines
     a. Load routines (direct, relocatable)
     b. Dump routines
     c. Device control routines
2. *Application independent*
   - File management
     a. Sort
     b. Merge
     c. File manipulations
   - Report generation
   - Data retrieval
   - General purpose management information systems
3. *Application dependent**
   - Billing
   - Accounting
   - Inventory control
   - Marketing analysis
   - Production control

*Less enthusiastically received by users, because these programs are generally inflexible to more than one tailored application.

8-8 ALLOCATION OF MEMORY DURING OPERATION

Generally, the individual program remains in main memory proportional to the amount of use. Some subprograms, e.g., loading the main memory from a magnetic tape, may remain in core constantly. Others in this classification could include much-used subroutines of all types, on-line diagnostics, and the executive program—including its relocatable loader. These are assigned an area (or sectors) in core and left pretty much undisturbed during the operation of the individual application programs.

The application programs are generally loaded into core as required and dumped from core when not in use—unless, of course, the memory is large enough to handle everything at one time. Other areas of core are set aside to store input or output queues (i.e., data being gathered in being readied for output). As an example, there can be an area set aside to hold data for a CRT display or an area where the printout page is formatted.

One other area is normally required. This is the *work area* (or scratch pad), where operations of all descriptions are performed on the data and perhaps the instructions.

In addition, in some machines certain processor registers such

as index registers, counters, and interrupt instructions are included in dedicated slots of memory. These slots are never used for any other purpose.

8-9 A LANGUAGE FOR TIME-SHARING—BASIC

8-9.1 General

This language permits the terminal operator to interact (put in commands and read out results) at a time-share teletype terminal. Because the time-share computer completes the action for each statement and may require operator response, the language is *not* called a compiler, but is known as an interpreter. Minicomputers effectively use BASIC, as a compiler.

Although conversational versions of many compiler languages exist for time-sharing operations, including *conversational* FORTRAN (discussed in Chapter 9), the occasional user of a time-sharing system does not usually need the power and sophistication of a professional compiler language. He wants a language that is effective but easy to learn and easy to use. Such a language is BASIC, developed originally at Dartmouth College for use on its time-sharing system. Because of space limitations here, only a few basic comments and brief examples are given. Also, as time advances, even this simple interpretive compiler is continually being upgraded in its limited capability. It is basically a simplified FORTRAN. The symbols used in BASIC consist generally of multiplication, division, addition, subtraction, and exponent manipulation. (Note the similarity to the FORTRAN cases.)

8.9.1.1 Types of Statements The types of statements allowed in BASIC are briefly tabulated in Table 8-4. This approach was used to present the basic material as compactly as possible. Other miscellaneous statements included in BASIC but not tabulated include:

REM Used to insert *remarks*.

DEF Used to define functions in a program.

RND Used to automatically generate random numbers.

GOSUB Used to enter a subroutine.

RETURN Used to return to statement following GOSUB.

RESTORE Used to reread data more than once in a program.

For control purposes, statements are provided that are typed by the operator while using the console. Table 8-4 gives these statements with their function

TABLE 8-4 Examples of BASIC statements

Statement type	*form*	*use*
LET	LET <variable> = expression	Assigns values to variables.
PRINT	PRINT expression, or message	. . . Print variable, constants, formulas, etc.
END	END	Last statement.
GO TO	GO TO <statement number>	Transfer control to statement designated.
READ	READ <variable list>	Reads in data; assigns first data variable to first data statement; etc.
DATA	DATA <variable list>	Defines data values.
INPUT	INPUT <variable list>	Inputs number from keyboard.
IF THEN	IF <expression> operator <expression> THEN <statement number>	Tests for an inequality, transfers control on the basis of test results; operators are < less than > greater than <= less than or equal to >= greater than or equal to = equal <> not equal
FOR	FOR <variable> = expression to expression step expression	Sets the control condition for a multilisting loop.
NEXT	NEXT <variable>	Returns loop to start.
DIM	DIM $a(a_1,a_2)$, $m(b_1, b_2)$	Defines dimensions of matrix n and m; a_1, a_2, b_1, b_2 are the respective dimensions of the two matrixes.
STOP	Stops operations at once	
RUN	Begins computation or program	
SCRATCH	Destroys present program being run	
LIST	Causes a listing output of the present program	
LISTxxx	Causes a list printout beginning at xxx	

Also, many mathematical functions are available to the programmer. These include:

SIN(x) sine of x
COS(x) cosine of x
TAN(x) tangent of x
ATN(x) arc tangent of x

These are samples, and there may be other expressions in a table of statement.

In addition, BASIC has a built-in function to manipulate matrices. These statements include such features as sizing the matrix, adding matrices, multiplying matrices, inverting matrices, and printing matrices.

The student is referred here to the more thorough references on BASIC for instructions on how to write and use these tools.

In any practical time-share system, there are variations on these statements, and either some may be left out or others added. The potential user must study the instructions given with a particular system to know its application and limitations.

8-9.2 Sample Program

In order to use the time-shared system from a remote console, the operator dials the computer by use of a special telephone number. The computer responds by typing a reply on his teletype asking for an identification. The operator types in the identification, and if it is a valid one, the computer replies READY. The operator now can run his program. If he is to write the program in BASIC, he types BASIC. The computer replies with a READY. Let us assume that the problem is to find the hypotenuse of a right triangle; he types out the following program:

```
10 LET X = 5
20 LET Y = 1 0
30 LET HI = SQR (x↑ 2 + Y↑ 2)
40 PRINT "THE HYPOTENUSE IS" HI
50 END
```

Step 50 completes the simple program insertion. To run the program the operator types RUN.

The computer then responds with

```
THE HYPOTENUSE IS 5.099
READY
```

The READY indicates that the computer is ready to receive a new instruction from the operator. If the operator is through, he types SCRATCH and OUT, which both erases the program and turns the system off. If he is not through, he can give the program other instructions instead, such as a new value of X and Y.

```
10 LET X = 10
20 LET Y = 10
   RUN
```

The computer replies

```
"THE HYPOTENUSE IS" 14.142
READY
```

This is a truly interactive system. The operator can modify and continue at a speed he sets for himself. By a SCRATCH of the original program, he could rewrite it so that the three values of the hypotenuse were printed:

```
10 READ X,Y
20 PRINT "THE HYPOTENUSE IS " HI
30 LET HI = SQR (X ↑ 2 + Y ↑ 2)
40 GO TO 10
50 DATA 1,1,1,5,3,4,100
60 IF X IS 100 THEN 70
70 END
```

The revision will cause a printout when run is typed:

```
"THE HYPOTENUSE IS" 1.414
"THE HYPOTENUSE IS" 5.099
"THE HYPOTENUSE IS" 5.000
READY →
```

This illustration should indicate the general procedure and the ease with which the beginner may program in BASIC. A more experienced programmer might prefer to use conversational FORTRAN or any of several available languages. Most time-sharing services offer a comprehensive language choice and operational programs suited to the special problems of their clientele. Many minicomputers supply BASIC employed primarily as a compiler, but still allow operator intervention as in a time-share program.

8-10 SUMMARY

The original method of writing programs in (1) and (0) code form has disappeared like the dinosaur. Today programs are usually written in a near-English form (a *compiler* language). A compiler software program then converts the *source program* to computer language to a (1) and (0) object code understandable to the machine's logic.

Machine programming is sometimes done in an intermediate form—*assembler* language. Here mnemonic names are substituted for both operation and address codes. The assembler program then performs the translation process from the symbolic language form of mnemonics to the machine-compatible code form of (1s) and (0s).

The remainder of the chapter discusses other aspects of software as well, defines program types, and concludes with a short explanation of the interpreter BASIC.

In assembler language programs, each machine language state-

ment is assimulated by one or more mnemonics. Thus, there is a one-to-one statement correlation between the program in the two forms (source and object).

A program written in a *compiler* or *interpreter* language form may cause a single compiler statement to generate many machine language statements. This then would be very efficient from the program-writing standpoint; however, compilers tend to inject superfluous code at times. There may be some operations not definable in the higher-level compiler language. Since individual instructions can be defined and specified in assembler language, assembler programs tend often to be more compact, but usually require more writing skill on the programmer's part.

The collection of programs, routines, etc., required to operate a computer and cause it to solve problems and document methods of problem solving effectively is defined to be software. Among the simpler routines are loading programs, which are necessary to load other software into memory; in fact, the first program, put in by hand, is a bootstrap loader that will load a more general loading program into memory.

Other software routines include those that operate the peripherals, organize the data, create and operate on files, perform a series of calculations, permit file data to be edited, check out the hardware, write and compile programs, check out software, perform operations, call subroutines, etc.

The more common general-purpose utility software is often included with the hardware purchase. The user, however, will have to procure or generate special programs to do his detailed jobs. If he is lucky, even these can be purchased from one or more software suppliers. Subroutines and subprograms are basically equivalent things, i.e., parcels associated with the main program.

8-11 PROBLEMS

1. Define the term *programming*.
2. Define *operational, executive, utility,* and *debugging* programs.
3. What is a *macroinstruction?*
4. Describe three types of *loading* programs.
5. Define several types of *subroutines.*
6. What are some of the software packages associated with business processing?
7. How is memory allocated within a computer?
8. Write a program in BASIC to compute the area of a circle (A =

πr^2). Assume that calculations are required for $r = 3$ in., $r = 6$ in., $r = 12$ in.

9. When is BASIC considered an *interpreter* and when a *compiler*?
10. Write a program flow chart using a decision element.

9

FORTRAN and Other Compilers

9-1 INTRODUCTION

As has been pointed out previously, *compilers* were developed to ease the longhand task of writing programs by letting the processor itself translate from a human recognizable language to the eventual machine language code. With passing time there has been considerable improvement in available compilers. These improvements include compilers with wider capabilities, compilers that are more efficient in their generation of machine codes, and compilers that are easier to use. However, the ideal has not been reached—i.e., a compiler that can be used universally for all applications and for all processors and by anyone with a minimum of training.

The compiler languages that have received the most attention include ALGOL, COBOL, and FORTRAN. ALGOL has attracted worldwide interest, having an international representation on the standardizing committees. COBOL (Common Business Oriented Language) was designed to assist with business programming and is characterized by language rules that allow writing such extremely descriptive statements, for example, as

1. MULTIPLY WAGES BY HOURS EQUAL PAY.
2. SUBTRACT TAXES FROM PAY.

FORTRAN (Formula Translation Compiler) is selected for discussion in this chapter because it is the widest used compiler in recent years and is representative of the average compiler. FORTRAN originally was designed for the IBM 704. It was modified and improved to FORTRAN II and later to FORTRAN IV. Although FORTRAN is supposed to be a standard compiler language, just about every processor using it has some significant deviations from the standard form. These variations are to account for individual machine hardware limitations or to add additional capabilities not available in the standard form. For instance,

time-sharing and control applications require processing operations not normally provided by standard FORTRAN. For example, IBM has a time-sharing oriented FORTRAN-type compiler called QUICKTRAN. Also, manufacturers of control equipment and systems have developed FORTRAN-type languages that handle the processing of test-point measurements, multiplex sequencing, and many such operations as these that are not part of standard FORTRAN. Neither of these pseudo-FORTRANs are standard. In fact, the problem with compilers is generally associated with the lack of standard. IBM has taken the lead with the development of a special compiler for their 360 processor line. This compiler, PL/1, has many of the characteristics to develop a single, universal compiler that serves a wide range of both business and scientific processing applications.

The previous chapter discussed the *compiler interpreter* BASIC in some detail. This particular *compiler interpreter* language is easy for the novice or the once-in-a-while programmer to learn and to apply to time-sharing system problems. The experienced programmer, however, may perfer a *conversational* FORTRAN because of previous FORTRAN experience and the fact that generally more can be accomplished per language statement than is possible with BASIC.

To write a FORTRAN program, the user must learn such elementary things as how to write numbers in the language, how to write and use the FORTRAN statement, and what limitations must be observed. The statements, numbers, etc. must be formatted in a precise manner, strictly according to the rules, or the translation of the user statements will not be properly interpreted. A misplaced (or omitted) comma or period can be disastrous. Therefore, care in writing is an absolute must. Mistakes such as those indicated above cause the compiler program not to run at all or cause completely idiotic results if it does run.

To properly learn FORTRAN requires careful study of all aspects and sufficient drilling in the fundamentals. This chapter merely introduces the subject, and the serious reader who wants to know the details can consult a FORTRAN textbook or enroll in an appropriate class.

The fundamental method to write numbers in FORTRAN is given in the following section. In order to write a program successfully, one has to learn to write FORTRAN statements as generally outlined in this chapter. These statement types are classified as

1. Input/Output
2. Arithmetic (or assignment)
3. Control
4. Subroutines
5. Specifications

Sections 9-2 through 9-7 discuss briefly the organization of these statements and their particular functions in the program. Some elementary examples are included.

9-2 NUMBER AND CONSTANT REPRESENTATION

In FORTRAN, variables and constants can be represented as fixed-point or floating-point numbers. The fixed-point quantities are restricted to integer values, can be signed or unsigned, and are usually restricted to a range of five decimal digits. Floating-point numbers are usually less constrained, allowing the decimal point to be placed anywhere within the allowed word length. A shorthand representation of a floating-point constant often used is to write the value coefficient followed by the exponent in the form 0.27 E-6, which equals 0.27×10^{-6}. Variables can be specified by one to six alphanumeric characters. Since there is no way to differentiate between fixed-point and floating-point variables, the program is made aware of what is meant by assigning the letters I, J, K, L, M, or N as the first character of a fixed-point variable name. The floating-point variable name can use any other letter as a first character, but not the ones listed for fixed-point variables above. There is a way to avoid this restriction, however, which is mentioned later in the chapter.

A formatted listing of variables in one, two, or three dimensions is called *arrays*. These are useful to represent matrices and other mathematical relationships.

The position of the individual subscripted variables within the array may be indicated by listing the variable symbol followed by one, two, or three subscripts. These subscripts are fixed-point quantities. A DIMENSION statement, discussed in Sec. 9-6, permits the program to allocate sufficient memory space to store the array quantities by specifying the extent of the array. Figure 9-1 demonstrates some examples of elementary arrays.

The important point in writing FORTRAN expressions is *not to mix fixed-point and floating-point numbers indiscriminately* in a given expression. If this is done inadvertently, a programming error usually results.

The next several sections of this chapter discuss the FORTRAN statement types. The order is arbitrary, with the simpler forms being covered in the earlier sections.

9-3 INPUT/OUTPUT STATEMENTS

The FORMAT, READ, WRITE, STOP, END, and PAUSE statements are elementary examples of input/output statements. A brief explanation of these statements follows.

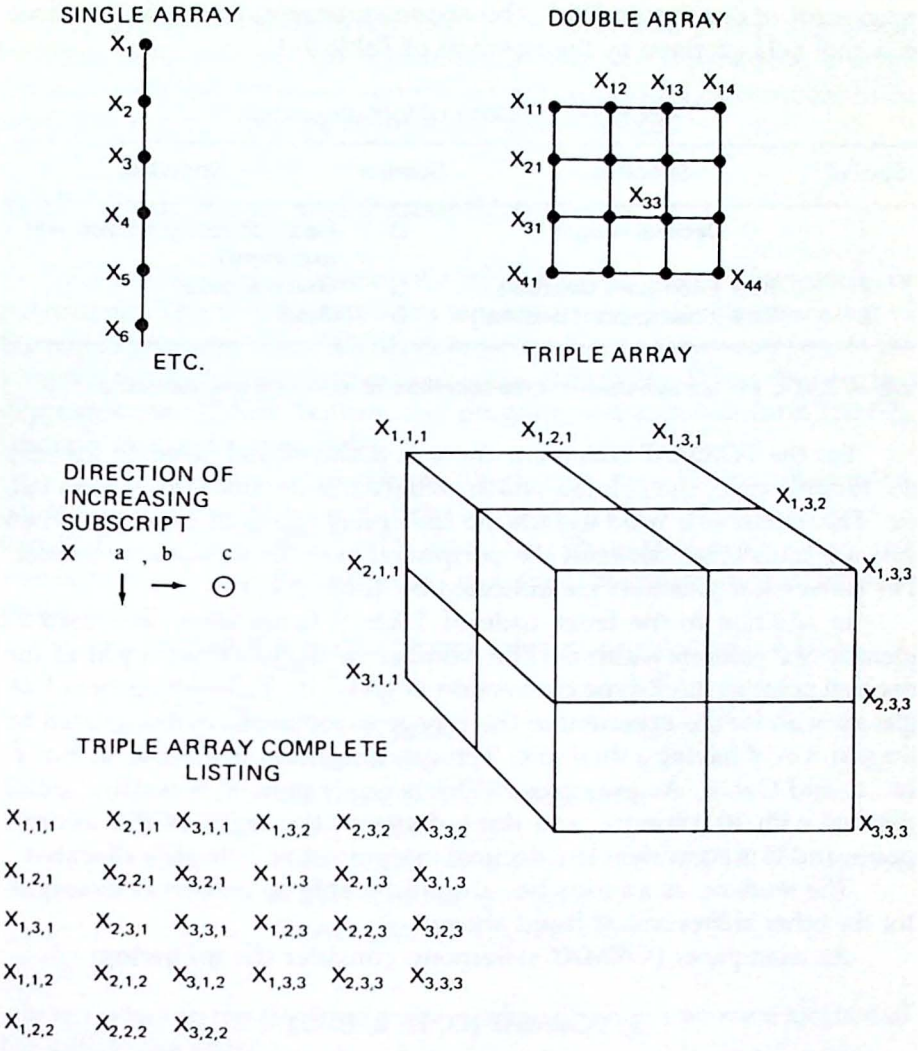


FIGURE 9-1. Example of arrays.

The FORMAT statement provides information to the compiler on how to interpret input data and put it in a form suitable for storage in processor memory. Likewise, it will take memory-stored data and arrange it in a form for the input/output display or external storage. The format statement is followed by a READ or WRITE statement to indicate the desired operation process. The FORMAT statement appears as

$$n \text{ FORMAT } (S_1, S_2, S_3, \ldots, S_m)$$

where n is is the statement number and $S_1 \ldots S_m$ is an ordered set of abbreviations, each one inserted by the programmer, to describe the ar-

rangement of one datum field. The above statement example describes m datum fields defined by the notation of Table 9-1.

TABLE 9-1 Listing of format codes

Symbol	*Specifies*	*Symbol*	*Specifies*
I	Decimal integer	D	Real (double precision with exponent)
F	Real (fixed-point decimal)	C	General data
E	Real (floating-point decimal)	O	Octal

Note. A, T, H, L, etc. are also used in some compilers to represent special items.

For the FORMAT statement there is a shorthand used to identify the variable type, the column width, where the decimal point is to fall, etc. This notation is used to indicate the conversions to be made when data are transferred between the peripheral and the internal processor. The conversion notations are indicated by Table 9-1.

In addition to the letter code of Table 9-1, numbers are used to identify the column width w. The number of digits to the right of the decimal point in the E-type conversion is given by d. The number of digits allowed for the exponent in the F-type expressions is designated by d again, i.e., d having a dual role. Typical designations are Ew. d, Fw. d, Iw. d, and Ow. d. As examples: E10.3 is equivalent to a floating-point decimal with 10 columns, and three digits to the right of the decimal point, and I5 is equivalent to a decimal integer—five columns allocated.

The student, as an exercise, should be able to construct examples for the other abbreviations listed above.

An example of FORMAT statement, consider the following:

25 FORMAT (I3, F3.2, E7.2)

where 25 is the statement number; I3 is the datum, is an integer three columns wide; F3.2 is the datum, is a real number without an exponent, three columns wide with two digits to the right of the decimal point; and E7.2 is the datum, is a real number in exponent form of seven columns, two to the right of the decimal point.

9-3.1 Read and Write Statements

These two types of statements are similar. The forms of the statements typically are

READ (i, n) list
WRITE (i, n) list

In the above statements the i is a peripheral device identifying number, the n refers to the statement number that is to be read, and the list is an ordered series of variable names, separated by commas, to be read. The n value may be omitted in some instances.

9-3.2 End, Stop, and Pause Statements

END is the last statement of a FORTRAN program, subprogram, or subroutine. The STOP statement is used to terminate the execution of the object program. the PAUSE statement allows a temporary stop to the program while the operator makes some adjustment. When the operator depresses the START button, the program will continue from PAUSE, where it stopped temporarily.

Further input/output statements that are used for control of external magnetic tape units, and disk files could include such statements as BACKSPACE, REWIND, END, FILE, etc. These statements are control instructions to the peripherals, and their function should be self-explanatory.

Other statements in the control category may include, e.g.,

```
PUNCH n list
PRINT n list
WRITE OUTPUT TAPE i, list
READ TAPE i, list
WRITE TAPE i, list
WRITE DRUM, i, j, list
READ DRUM, i. j, list
```

where i refers to the device number and j is the first statement location of the initial data entry.

As an example of the use of some of these statements one might write a simple program as follows:

C PROGRAM TO COMPUTE
AREA OF TRIANGLE

```
                                    Comment
1  FORMAT (F5.2, F5.2)              Reads values of B
2  READ (5,1) B,H                   and H from input tape.
3  A = 0.5/B /*H                    Computes A.
4  FORMAT (F5.2)                    Write output A on
5  WRITE (2,3) A                    printer.
6  STOP
7  END                              Ends program.
```

All the statements are explained in the text with the exception of Statement 3. This particular statement performs the calculation Area = $\frac{1}{2}bh$ and is of the type covered by Sec. 9-4. The first statement of the program above is a comment—indicated by the C at the beginning. The statement is printed on the printer readout, but it does not enter in any other way into the control or the problem solution. Comments are used to provide printed identification of a program for the operator's convenience in identifying the printout at a later date.

9-4 ARITHMETIC (OR ASSIGNED) STATEMENTS

The purpose of an arithmetic expression is to permit the processor to perform a calculation and obtain a numerical value for the expressions involved. As in ordinary arithmetic, there is a group of operational symbols that the computer recognizes. These are customarily defined as follows:

\+ results in addition, as A + B
− results in a subtraction, and A − B
/ results in a division, as A/B
* results in a multiplication, as A * B
** results in a raise to a power n, as A ** n

In statements having more than one of the above symbols the operations are interpreted by the computer in the following order: **, *, /, +, −.

To avoid confusion to the interpreting computer, parentheses are used to identify groupings of quantities as in ordinary arithmetic. As an example, the FORTRAN statement

y =(A+B**3) (A+B*C)

might be interpreted as number of ways (some erroneous perhaps), but (A − B** 3)/(A + B* C) is clearly

$$y = \frac{(A - B^3)}{(A + BC)}$$

In general, arrangements like A + − B are not allowed, but A + (−B) would be valid.

Because FORTRAN can perform arithmetic in several modes, it is necessary to pay careful heed to the way numbers are represented and

used. For instance, an integer (a whole number with no decimal point) must be represented in statements with names beginning with letters I, J, K, L, M, and N only. For real numbers (having a decimal point) or for numbers represented in exponential form the name must begin with *any letter but* I, J, K, L, M, and N. If this rule is not followed, the computer program will misinterpret the number representation, and chaos usually results. A possible exception follows in the next paragraph.

The above designation of number form is termed *implicit representation* and is the usual method of number representation in FORTRAN. However, there is a method to avoid or negate this restriction. The method involves the use of explicit-type statements.

The purpose of the *explicit* statement is to specify a value to be applied to a variable name and to define the variable type precisely. Examples of explicit statement are

```
INTEGER A, B, C . . .
REAL C, D, E . . .
DOUBLE PRECISION G, F, A . . .
COMPLEX S, T, P . . .
LOGICAL U, V, W . . .
```

The alphabetical letters assigned above are variable names, subprogram names, array designations, etc. The variable designations, a part of the above statements, when used properly, allow the programmer to deviate from the implicit rules discussed in Sec. 9-2.

9-5 CONTROL STATEMENTS

These statements control the operation of the program. Among other things, they allow the program to change direction (branch) depending on prescribed factors—i.e., the program may loop, skip, or implement decision results depending on conditions specified by the control statements. Thse statements include the GO TO, the logic IF, the logic DO statements, and a few others.

9-5.1 GO TO Statement

The GO TO statement in its most simple form allows the program routine to be transferred to another specific statement. An example would be

```
GO TO n
```

where the program is transferred to statement number n. This statement allows loops to be executed. At the completion of an operation, involving several statements, the GO TO statement leads back to an initial statement of the loop operation. The statement group then may peform the same calculation again, but on new (fresh) data. The GO TO statement can be written in a more complex form and will refer to many statements in sequence up to the value of an index k. As an example,

GO TO (7, 5, 9) k

means GO TO statement 7 when k = 1, the statement 5 when k = 2, and to statement 9 when k = 3.

9-5.2 IF Statement

The IF statement is a decision statement that allows the program to transfer to one of two designated statements depending on the evaluation of an expression. The elementary form is, if (A qualifier) n, where *A* is the logical expression to be evaluated and is one of the operations indicated in Table 9-2.

TABLE 9-2 "IF" Statement Qualifiers

Math Symbol(s)	*FORTRAN Abbreviation*	*Meaning*
$A > B$	*(A.GT.B)*	*A* greater than *B*
$A = B$	(A.EQ.B)	*A* equals *B*
$A \geq B$	(A.GE.B)	*A* greater than or equal to *B*
$A \leq B$	*(A.LE.B)*	*A* less than or equal to *B*
$A = B$	(A.NE.B)	*A* not equal to *B*
$A < B$	(A.LT.B)	*A* less than *B*

There are others, but these should indicate the general form of processing steps that can be executed.

The s represents the statement to execute if the logical evaluator is true. If the qualifier statement (s) is not true (i.e., false), the statement following the IF statement is to be executed. The symbol s can represent a statement number, it can direct a calculation, or it can designate an operation like WRITE, READ, PAUSE, etc. Examples are

IF (A.LE.B) 32
IF (X + Y ** 2 = 4.5) PAUSE

Other forms of the IF statement are possible. For example: IF (a)n_1 n_2, n_3, ...

The symbol *A* can be designated as an arithmetic expression and according to the results of a computation—whether the results are positive, equal, or negative—the transfer is to one of three other statements, n_1, n_2, or n_3, respectively, where n_1 is the statement number to which the transfer is directed. For example,

IF(X ** 2 − 4XY = 5) 21, 27, 32

indicates a positive result which requires execution of statement 21, an equal result ($X^2 - 4XY = 5$) requires statement 27, and a negative result requires statement 32.

In some cases one type of control statement implies the existence of other control statements. For example, program control requires the use of both ASSIGN statements and GO TO statements together. The ASSIGN statement effectively assigns a statement number to a variable name such as

ASSIGN n to i

where n is a statement number and i is the name for an uninscripted variable. The ASSIGN statement, however, is not operable unless there is a corresponding GO TO statement that also contains n and i.

The general form of the GO TO statement in this case is

GO TO i (n_1, n_2, n_3, . . . , n_m)

where i is the indexed variable and n_1, n_2, . . . , n_m are the statements to which control is to be transferred.

Several ASSIGN statements will normally reference a single GO TO statement. For example,

```
.
.
.
6 ASSIGN 12 to X
7 ASSIGN 14 to Y
8 ASSIGN 16 to Z
9 GO TO X (12, 19, 21)
```

Another powerful control statement form is the DO statement. This statement allows the loop iterations to be controlled. The purpose of the DO statement is to precisely define the following:

1. The sequence of statements to be involved in the iterated subprogram.
2. Define a variable initially to some value.
3. Increment the variable a specified amount.
4. Test to determine if the specified number of iterations has been performed.

The general form of the DO statement is

$$\text{DO n i} = m_1, m_2, m_3$$

The "DO n" term indicates do all the following statements down to statement n. At statement n the control is returned to the DO statement again. The i defines the unsubscripted variable to be exeucted beginning with the initial value of m_1. i is incremented at each step by its value of m_3. The process stops when a value of i equal to m_2 is exceeded and control passes to the next executable statement beyond n. As an example, consider

$$\text{DO 13 X} = 10, 20, 2$$

This translates to DO all statements up to statement 13, where X initially is equal to 10, until X = 20 is exceeded. The incremented value to be applied to X is 2. If the 2 were omitted, the increment would be interpreted by the program to be equal to 1.

Thus the DO statement in conjunction with the IF and GO TO statements provides a wide range of control to the program. It is possible and permissible to design loop within loops, provided that the inner loop is completely within the outer loop. To transfer control back and forth in and out of a DO loop, if not done properly, leads to programming troubles. It is permissible to transfer control from a statement within a DO loop to outside the loop, but not vice versa, i.e., to transfer into a DO loop. The reader is referred to a detailed FORTRAN text for further information about these procedures.

The range of DO is not permitted to end with a transfer statement, although a transfer may be required. To circumvent this possible dilemma, the CONTINUE statement can be employed at this point in the program. Thus the statement is a dummy statement designed merely to

end the DO range properly and to permit the program to continue on.

A simple program using many of these statements is given in Table 9-3. A student should be able to interpret the statements by comparing them with the explanation in the text.

TABLE 9-3 Simple FORTRAN Program Example

	C-Compute and print value of Z	*Comment*
	1 FORMAT (F6.2, F6.2, F5.2, F5.2)	Input format for *A, B, X, Y*
	2 FORMAT (F6.2)	Output format for *Z*
	3 READ (1, 2) A, B, X, Y	
	4 DIMENSION Z (20)	Put value of *Z* in array.
Loop	5 Z = (A ** 2 + B ** 2) ** 2/Y	$Z = \frac{(A^2 + B^2)^2}{Y}$
	6 T = 100.25	
	7 IF (Z, LET) GO TO 3	Statements 3-7. Loop Until Z is greater than 100.25.
100.25.		
	8 GO TO 13	Jump to print routine
	.	
	.	Print routine
	.	
	13 WRITE (2,3)Z	Printout of *Z* array values.
	.	
	.	
	.	
	STOP	
	END	

9-6 SPECIFICATION STATEMENTS

There are four statements that are classified as specification statements. These include the FREQUENCY, EQUIVALENCE, COMMON, and DIMENSION statements.

The first three of these are employed to improve the efficiency of the source program, and, as such, are never required, but can be considered optional with the user. The FREQUENCY statement is employed to indicate the number of times the programmer believes each branch specified will be executed during the course of the program. It is written in the following manner:

FREQUENCY M (i, j), n (k, 1) . . .

where n and m are statement numbers and i, j, k, and l are unsigned fixed-point constants.

The EQUIVALENCE statement permits the option of allocation of data storage in the object program. Two or more variables are permitted to share the same memory locations during the object program execution. The statement is in the form of:

EQUIVALENCE (a, b, c, d) (d, e, f, . . .)

where a, b, c, etc. are individual variables and can be followed by a single subscript.

The COMMON statement allows two or more program statement sets, data, etc., to be stored in the same memory area apart from the main program storage area. This group of data, array, subroutines, etc. can be shared between two or more programs or subroutines. The statement form is

COMMON A, B . . .

where A, B are variables, nonscripted array terms, or subroutine names, as typical examples.

The FORTRAN IV language permits the manipulation of numbers and variables arranged in array form. The statement used to define these arrays are called DIMENSION statements. Arrays may be of one, two, three, or more dimensions. The variables appearing in the arrays must be given defined values by other statements in the program.

As an example, the statement

DIMENSION B (10)

refers to the variable B arranged in a one-dimensional array of 10 terms. Another example might be:

DIMENSION U (6), V (3, 3) W (2, 2, 2)

The k (i, j, K) terms following the word DIMENSION define the individual terms of the arrays to be as follows:

U Array	*Two-dimensional V Array*	*Three-dimensional W Array*
U (1)	V (1, 1)	W (1, 1, 1)
U (2)	V (1, 2)	W(1, 1, 2)

U (3)	V (1, 3)	W (1, 2, 1)
U (4)	V (2, 1)	W (1, 2, 2)
U (5)	V (2, 2)	W (2, 1, 1)
U (6)	V (2, 3)	W (2, 1, 2)
	V (3, 1)	W (2, 2, 1)
	V (3, 2)	W (2, 2, 2)
	V (3, 3)	

Thus three separate array formats, each of a different dimension type, are defined by a single DIMENSION statement. The form of these arrays is illustrated by Fig. 9-2.

The program assigns variable values to the subscripted variables of the array as instructed by other statements in the program. Because of the need to limit this discussion, again the reader is referred to established FORTRAN texts for a complete description and examples of the numerous array manipulations possible.

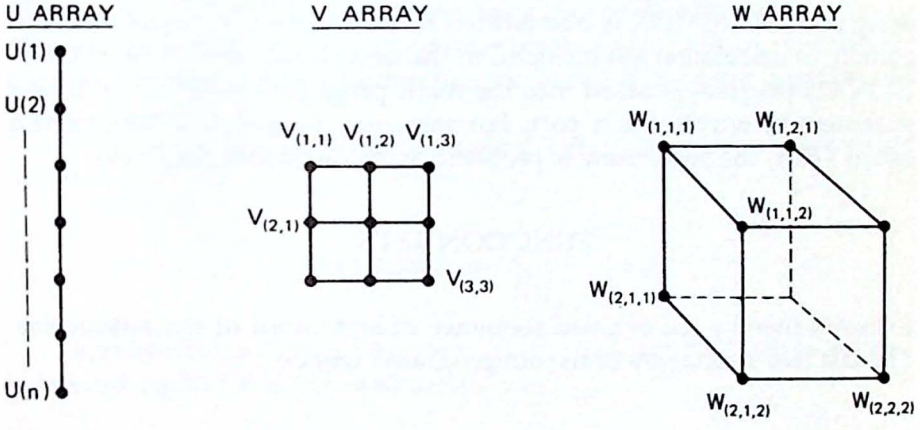


FIGURE 9-2. Designated array forms.

9-7 SUBPROGRAM STATEMENTS

Often a particular arithmetic operation involves a defined sequence of operations such as finding the sine of a number, the square root of a number, its natural logarithm, or some other function where the computation involves a series of computing steps or lookups in tables. Such subprograms are inserted into the main program merely by writing the statement and identifying the variables to be operated upon. These subprograms are called *predefined subprograms*. They may be incorporated as part of an arithmetic statement.

Examples of such subprograms include, but are not limited to:

SINE

COSINE
ARC TANGENT
SQUARE ROOT
NATURAL LOG

One might incorporate these into a program as follows by simply writing:

```
Y = 4.0 SIN (X) + SQRT (X + 1)
```

This would compute, of course, the arithmetic value of

$$Y = 4 \sin X + \sqrt{X + 1}$$

for tabulated values of X.

Not all subprograms are of the *predetermined* type. The type designated FUNCTION is one written to accomplish a special task, operation, or calculation not included in the first classification. The FUNCTION subprogram is called into the main program by simply writing a statement of which it is a part. For instance, to define a subprogram called TETS, the subroutine is prepared with the initial statement

```
FUNCTION TETS
```

followed then by the detailed sequence of statement of the subroutine. The last two statements of the subprograms will be

```
RETURN
END
```

These latter statements order a return to the main program, signaling that this is the END of the subprogram.

Two examples of how the main program calls a subprogram by including the subprogram name as part of a statement could be represented by the statement

```
IF (TI.GT.10.21) GO TO TETS
```

or

```
READ (5,6) X, Y, TETS
```

Also, for the convenience of conserving memory, a central location in memory may be used to store values commonly used in more than one subroutine or subprogram. Each program then must contain a COMMON statement such as discussed in Sec. 9-6 to indicate its need to employ this subprogram value on a shared basis.

Subroutines are like subprograms but do not necessarily transfer the results of a computation on to the main program at the completion of their execution. There are other subtle differences, fully explained in a FORTRAN text, but for purposes here the subroutine is defined by the statement

SUBROUTINE name (a_1, a_2, a_3, . . . , a_n)

Name, of course, is the name of the subroutine and a_1, . . . , a_n are the argument list, which may include variable names, array names, dummy names, etc., and include arguments to be passed back and forth between the main program and the subroutine itself. There need be no entry for argument if none are to be interchanged between the main program and the subroutine.

A subroutine is called to the main program by a CALL statement of the form

CALL name (a_1, a_2, . . . , a_n)

The functions defined by name, a_1, . . . , a_n are the same as defined before the SUBROUTINE statement.

9-8 CONVERSATIONAL FORTRAN

The statement and procedures presented so far have applied to FORTRAN as used in a *batch* system. As defined previously, a batch system is one where the program is completely written in advance in its entirety and loaded in the processor as a unit. Likewise, the results are obtained as a single printout—or other unit form of output.

Time-shared systems, discussed in Chapter 1, are different in that the operator sits in front of a terminal (teletype keyboard or CRT display) and puts his program initially on the remote machine either by typing the individual program statements or by punching up a paper tape in advance and inserting this in a paper tape reader, for example.

In any case, the results of the computation are read back to the operator directly via the terminal output device. If he is satisfied, fine, but the chances are that he will want to explore further. It is a great con-

venience, in fact the prime advantage of time-shared systems, that the operator can almost immediately change the input data and the program itself by use of his keyboard. By typing a single FORTRAN-like statement, he can cause the program to test itself and report the result of the test back via the teletype. He looks at it, decides what statements must be modified, modifies them, and the new modified program is ready to run, if, of course, the modification was performed properly.

Because of the need for the operator to interact with his program, a special FORTRAN is often provided, usually called *conversational FORTRAN*. This language is similar to the batch-oriented FORTRAN but has a number of additional statements for controlling and interacting with the program as indicated here. A few typical ones are listed for illustration in Table 9-4. In Chapter 8 a short resume of the language BASIC was presented. This is an example of another conversational language that is somewhat less sophisticated than *conversational FORTRAN*, and can be used in a interpretive mode (see Sec. 8-9.1).

However, operators trained in FORTRAN usually prefer to work programs in this language, when it is available; hence, this is strong impetus for time-shared operations to include *conversational FORTRAN* as an interpreter language.

TABLE 9-4 Typical Conversational FORTRAN Statements

Statement	*Explanation*
EXECUTE	To run program.
KILL	To take the complete program out of the computer.
DELETE n	Delete statement n.
DISPLAY n	Print out value of n.
LIST	Print out program for user examination.
RESEQUENCE	To change statement number.
COMPILE	Insert or modify text.
ACCEPT A, B, C	Enter A, B, and C into program.
SAVE n	To save a program file n.
EDIT n	Print statement n so that user may edit using control characters.

9-9 LIMITATIONS TO FORTRAN

The language of FORTRAN has continuously been improved since its introduction in the middle 1950s. FORTRAN applies specifically to scientific, engineering, or mathematical problems and is not especially useful for problems of a business nature. It has powerful calculating capabilities. ANS1 Standard FORTRAN IV is available on practically all mini, medium, and large- scale computers. Individual programmers

often modify the language to suit their specific applications. One modification, of course, led to *conversational FORTRAN*. Other work has led to the adoption of FORTRAN for such applications as real-time gathering of multiplexed analog sensor data, and remote control of output controllers. Although these adoptions are not standard FORTRAN IV, it is likely that in the future they could well be. The language is continually being updated (although not necessarily recognized as standard) to include:

1. Special I/O statements to provide nonbuffered outputs to peripherals.
2. Symbolic on-line debugging options to observe programs actually in operation.
3. Operating system command subroutines which provide a capability to users to write parts of programs in nonstandard languages (to FORTRAN) and yet have these called into the main program.

There are many other improvements, but these are examples of some included in General Electric's Mod FORTRAN FIV. In general, for specific real-time applications, FORTRAN execution may be too slow or specialized. This problem leads to the development of new and more specialized languages, such as HAL-5 for the space shuttle flight control application. Usually a program written in FORTRAN is not as compact in use of memory storage as one written in the machine's assembly language, nor will it execute as rapidly.

9-10 COBOL

This section attempts to review very superfically the general aspects of COBOL (an acronym for Common Business Oriented Language, which pretty much explains its application). The language is intended as a means for direct incorporation of a business program to a computer employing a suitable COBOL compiler. The language employed in COBOL is a collection of English statements, which provide, when prepared correctly, instructions to perform business-related problems in a relatively machine-independent manner on a computer.

The COBOL language is divided into four major parts called *divisions*. A programmer can write into the proper divisions a complete description of the data, procedure, and equipment required to run the problem. The divisions are as follows:

1. **Identification Division.** Identifies the source problem, includ-

ing program name, author, date written, data compiled, installation identification, security, and remarks.

2. Environment Division. Specifies the computer equipment to be employed in both compiling and object program runs. Information relating to input/output control and special hardware equipment characteristics can be included. This division divides itself naturally into two sections:

(a) The configuration section. Relates respectively to the characteristics of both the source and object computers.

(b) The input/output section. Deals with the handling of data transmissions between the object program and external mediums. The reader should refer to operational manuals before attempting to write (or rewrite) this program section if different computers are to be employed rather than one where the original program was run. Although COBOL is supposed to be machine-independent, with different hardware configurations this is very seldom the case.

3. Data Division. This division describes data that are to be accepted as input, manipulated, created, and supplied as output. There are two categories for this:

(a) Those data which enter and leave the computer as files from and to special areas.

(b) Those data which are created internally, placed into temporary or working storage, and/or formatted for output reports.

The data division is subdivided into the following:

(a) The file section. Where data contents are stored specifically in cards, tape, disk, or what have you.

(b) The working storage. Describes internal files and records not directly visible as input or output, but created temporarily internally during processing.

(c) The constant section. Where nonchanging items from the source program are stored.

(d) The report section. Where a description of the format and contents of externally generated reports are stored.

4. Procedure Division. This division describes the processing steps to be applied to the data. The procedure can be written in En-

glish prose, but careful attention must be paid to the use of COBOL assigned words, for example:

(a) Imperative statements. Specifies action to be taken by the object program, such as:

PERFORM	ACCEPT
PROCESS	ADD
SEEK	COMPUTE
SORT	MOVE

(b) Conditional statements. Specifies specific conditions or actions to be taken because of a test:

IF	RETURN
SEARCH	

(c) Compiler directed statements. Directs COBOL to perform specific actions at compiler time:

ENTER	NOTE
INCLUDE	

There is also a COBOL library which lists text for the environmental, data, and procedure divisions.

A typical COBOL statement could be:

SUBTRACT CY FROM ROLL.

In this case, the program is directed to subtract a quantity (or file) CY from a quantity (or file) from ROLL. There would be other statements in the Data Division expecially, but, perhaps, in other divisions as well that would direct the program to CY and ROLL.

Since the subject of COBOL language is reasonably complicated, it can hardly be treated in a few paragraphs. For further study, the reader is directed to other references, such as Awad and Crawford, as well as the COBOL Standards, and to appropriate manuals from computers having COBOL compilers.

As a summary of the features of COBOL there are

Sorting. The sort feature permits the user to express special sorting features often required for processing large data files, such as additions, deletions, creation, alteration, editing, or other file record modifications. There are available a variety of sorting procedures for the user's choice.

Report Writing. With this feature, the user initiates, generates, and terminates reports. He describes his report in the DATA DIVISION pictorially, to minimize PROCEDURE DIVISION coding.

Table Handling. This feature permits the creation, extraction, and other manipulation of tabular data. Another technique called *indexing* eases table searching and manipulation of specific items.

Mass storage and access. This feature accesses mass storage devices. Environment Division statements allow the user to express his mass storage files, allowing either random or sequential access.

Segmentation. Segmentation provides users the capability to subdivide Procedure Division statements. These statements, during operation, permit overlaps or link proceedings. Segmentation is provided in three types: (1) fixed permanent segments, (2) fixed overlap segments, and (3) independent segments.

Modularization. Whereas segmentation deals only with the Procedure and Environmental Divisions, modularization separates data processing programs into specific modules. Each module is then developed and debugged separately. The loader program then links these together and permits modules of large programs to overlap each other in a limited amount of core.

Other features of COBOL include:

1. Multiple use of data throughout many COBOL programs.
2. COBOL library. Permits a user to create a library file to avoid lengthy repetitions of data and procedures.
3. Error detection and correction. COBOL permits listings, error analyses, and powerful diagnostics. Error debugging program can be written in source language rather than reliance on core, disk, or tape dumps to trace error.

9-11 SUMMARY

This chapter summarizes to some extent the principal qualities of the FORTRAN *compiler*. It is by no means a complete picture as provided in those texts emphasising the language, or in FORTRAN compiler manuals employed with the machines to be programmed. Another compiler, COBOL, is sketched even more lightly, and again the student should refer to more extensive treatments in specialized texts or manuals. The object here has been to show their general usage and relative complexities, so the student at least knows where to start his study of programming languages of higher level than *assemblers*. Programming groups all over the world are constantly changing these compilers and developing new ones, so it is difficult to predict what modifications to these compilers will be standard next year. Programs do not remain con-

stant as originally written. There are always changes to make, so sufficient records must be made and kept up to date on what a program does, how it works, and its limitations.

Besides the FORTRAN and COBOL compilers, as well as the BASIC interpreter, there are newer ones that are becoming important. One of these developed by IBM shortly before 1970 is PL/1, which combines the features of both FORTRAN and COBOL in a single compiler. This particular compiler appealed so to the microcomputer supplier INTEL that they developed a special version for use with their line consisting of the Models 8008 and 8080 at the time. Their compiler PL/M, has been copied by just about every other microcomputer supplier, including Motorola and National Semiconductor, to name two.

9-12 PROBLEMS

1. What is the chief reason that programs are written in compiler form?
2. Write the floating-point number +.0000267 in FORTRAN notation.
3. Sketch a three-dimensional array with 27 elements.
4. What is a *format* statement? Give an example.
5. Explain the statement: READ (2, 25) A, B, C.
6. Write a program to compute the area of a rectangle ($A = bh$). List the name of the program at the start.
7. What is a GO TO statement?
8. Explain GO TO (1, 10, 15, 13)K where K = 4
9. What is conversational FORTRAN? To what other language is it similar?
10. COBOL is most used in what types of applications?

10

Minicomputers (First Generation)

10-1 INTRODUCTION

Minicomputers appeared on the market about 1968 pioneered by Digital Equipment Corp.'s (DEC) model PDP-8. This is a minimal computer designed for simple control applications. Shortly afterwards several minis were marketed by several organizations, including improved models by DEC. The design of minis required the development of ICs mounted on pull-out cards. Many manufacturers adopted automated wire-wrapping of chassis connectors. This chapter discusses a typical mini with which the author is somewhat familiar.

10-2 CHARACTERISTICS OF MINICOMPUTERS

Although there are individually wide variations between individual model types built by the same and competing manufacturers, most units will possess many of the following characteristics.

10-2.1 Price Range

This will be at the low end—usually less than $30,000. Many models are in the $2000 to $5000 range. Using microprocessor elements, the prices are now (1979) in the hundred dollar range.

10-2.2 Organization

This will include a parallel organization, binary 2's complement arithmetic and a word length of usually 8, 12, or 16 bits. Hardware multiply and divide is not always included, but usually can be added as an option. Memory is modular in 1K*, 2K*, or 4K* increments. The mem-

*K equals 1024 words.

ory cycle time is usually from 0.7 to 4 μ-sec for machines employing core memories. The newer semiconductor memories are also in wide use, and as a consequence memory access times will be higher than for cores, the precise access times depending on exactly the semiconductor memory type.

10-2.3 Instruction Words

With 16 bits the addressing capability may be limited to 1K words or less. Six bits of these are likely to be used in op code or variant bits, leaving only 10 for addressing memory. This means that to address memory outside of the 1K sector requires special addressing techniques—that is, the present instruction address must be modified. Instruction words for the H316 computer are discussed in Chapter 7, including the other categories of instruction words.

10-2.4 Software

10-2.4.1 General. An extensive amount of manufacturer-developed software is normally supplied in the form of paper tapes, card decks, magnetic tapes, etc., depending on the system peripherals purchased. Also, an assembler listing and written instruction for the use of each program will probably be supplied. Included among such software will most likely be:

1. Loading programs. Probably both an *absolute loader*, which specifies exactly where a source program loads into memory, and a *linking loader*, where a source program loads into memory as specified by the *relocatable assembler*, but not necessarily in the absolute addresses specified by an absolute loader. A linking loader is a valuable aide to moving programs back and forth between main memory and disk or tape file storage. It fits a program in available space.
2. Diagnostic programs. These are programs to assist in finding faults in either the hardware or software.
3. Assembler or compiler. These are programs to assist in writing other software. (See Chapters 7, 8, and 9.)
4. Utility programs. These include programs, or merely subroutines, to operate peripherals and provide mathematical functions.

10-2.4.2 Utility routines The demand for certain subprograms is

so great that the manufacturer develops them on his own. Such programs include:

1. Arithmetic processes like multiplication and division (if the machine does not have hardware for these processes).
2. Double-precision or floating-point operations.
3. Trignometric functions, logarithmic functions, exponential functions, etc., as well as other mathematical techniques.
4. Peripheral read and write routines.

Utility programs constitute a large portion of those referenced by the application programs developed for a particular system. They can be called into the main program as any other subroutine.

10-2.4.3 Compilers and interpreters Minicomputers of sufficent memory size now provide some compilers of the FORTRAN, ALGOL, or COBOL, types and probably an interpreter of the BASIC type. Programs written in these languages may not use memory as efficiently as the *assembly* language, or they may not be able to support real-time applications such as a multiplicity of A/D inputs or D/A outputs, but a compiler form is usually easier to write than an assembler form. Interpreters generally execute at a slower rate than compilers, since each interpreter instruction executes as it enters the machine. However, this allows the operator to interact with the rpogram, thus permitting him to control the program flow to a large extent.

10-2.4.4 Executive This program (sometimes called the *supervisor* or *monitor*) coordinates the execution of all other programs by providing such functions as job scheduling, storage allocation, and monitoring of operation of all other programs. It also calls up the necessary peripheral subroutines.

Manuals supplied include: *Programmer's reference manual, user's guide, interface manual, installation manual, parts lists, teletype manual,* manuals pertinent to each peripheral or option supplied, and software descriptive material. Without any software at all, but having at hand a programmer's reference manual, an operator can key in a short program by employing switches on the control panel console. Not only can he enter a program, but he can check it out also. Such an approach is hardly sufficient for the average application program. It is much more convenient to initially load an assembler or compiler program and follow this by loading (via an appropriate peripheral) the source program to be converted to machine code. After the conversion process, a program listing

should be typed out on the teletype in either octal or mnemonic code. This affords a chance to examine what is actually in the machine's memory, and allows correction to be made if necessary.

A very easy-to-use program called PATCH exists, and allows a nonprogrammer to type in via the teletype in either mnemonic or octal form short programs. These are entered, edited, or listed by using interpretive commands typed on the keyboard. This program aids greatly in the design of special interfaces, permitting the designer to type in his own PCU test routine.

To load any program initially the processor requires a proper loading program. This utility program has to be tailored to the peripheral being employed. It is also preceded by a short *bootstrap load* program of approximately 15 instructions entered manually in sequence via the control panel switches. This *bootstrap loader* then allows the primary load program followed by the application program to continue its loading mode.

10-2.5 Fabrication

The mini is usually built up from plug-in cards or modules, all pretty much self-contained in a sheet metal enclosure that can mount in a standard laboratory rack or on a stand. ICs and other electrical components mount on the individual cards and are connected by etched metal paths. The plugs for the cards will often be wire-wrapped, a technique that leads to automated wiring assembly. For troubleshooting purposes the metal enclosure readily opens, allowing cards to be replaced with known good ones. A control panel permits the operator to inspect the condition (location) of the instruction counter, observe the contents of important registers, or inspect the contents of addressed memory cells, and, if need be, rewrite the contents. There are other features that assist the operator to troubleshoot his program; for example, the instruction counter may be advanced by hand one step at a time; thus, the contents of successive memory cells are visible.

For peripherals selected as members of the product line, the interface PCU cards (or modules) are included in the basic price of the peripheral. The Honeywell system adopted for the DDP516 is to bolt into the main frame a μ-block containing up to 24 card slots. Three μ-blocks will bolt into a H316. Additional μ-blocks require an additional chassis. For the more common peripherals, such as a teletype and high-speed paper-tape reader, the PCUs may be a card similar in general appearance to the standard processor cards. The one peripheral indispensable for computer operation is the ASR teletype, either model 33 or 35.

10-2.6 Limited Service Provided by the Manufacturer

Because of low profit margin the manufacturer does not supply extensive service free of charge. In one case the service supplied merely involves the initial installation and checkout by running three diagnostic tapes. These would include:

1. A checkout program for memory
2. A test program for the CPU
3. A checkout program for the teletype

On the basis of these programs it is determined if the processor operates according to specification.

10-3 ORGANIZATION EXAMPLE

10-3.1 H316 Computer

The H316 is a minicomputer designed for real-time system applications involving on-line control, data logging, data formatting, process monitoring, and message switching. A powerful instruction repertoire (72 instructions) implements these applications, along with a moderately fast memory cycle time of 1.92 μ-sec, and an I/O data-handling rate of 130 K words per second with the use of a DMC option. The H316 has half the speed of the DDP516, but it is less costly, employing a standard line of ICs. It is also a much smaller physical package. The computer employs a stored program, which is a parallel binary type using a two's complement machine code.

A single 10-bit address with indexing and indirect addressing accesses a 16-bit core memory. A single hardware index register provides a limited indexing capability. Core memory size is from 4K to 16K, and is optionally expandable to 32K. The 72-instruction repertoire includes data transfer, byte manipulation, I/O control, and arithmetic operations, as well as a variety of logical, shift, and inducing operations. A high-speed arithmetic option (multiply in 10.56 μsec and divide in 21.12 μsec maximum) is available by adding 10 instructions to the H316 repertoire.

Including instruction fetching and execution, most internal operations require 3.84 μsec or less. A sector addressing technique permits the majority of instructions to be one word. Up to 1K words can be addressed directly by a single memory reference instruction. I/O operations can occur in one of three modes (direct I/O with or without priority interrupt, or DMC—in either normal or high-speed modes). The standard interrupt is a single line accessed by all peripherals involved.

10-3.2 Internal Organization

Figure 10-1 is a simplified block diagram of either the H316 or the DDP516 computers. The distribution control logic includes a central register (D-register) through which all data flow. Entry into memory is via the M-register which is possible either through the summing network (adder) to the D-register, or from the adder directly to the M-register. The M-register is, of course, our old friend the MBR register. Adder arithmetic operations are in two's complement code, which avoids the necessity to consider sign, except to note overflow or underflow.

All registers shown in Fig. 10-1 have 16-bit capacities except the F-register. This register has a four-bit capacity and copies op code established by bits 3 through 6 of the M-register. The A-register is the primary accumulator and communicates via the I/O data buses with the outside world. The B-register holds the less significant portion of arithmetic operands in excess of one word length. The P-register contains the location of the next instruction. Its contents are incremented by 1 each time an instruction executes and may be incremented an additional number of times during the processing of certain commands. For a jump instruction, the P-register is loaded with the memory location to which the program is to jump. During a compare and skip instruction, the P-register can be loaded to skip one or two instructions.

The Y-register is the 16-bit MAR. The X-register is a 16-bit index register used for address modification. Some of the registers (such as the

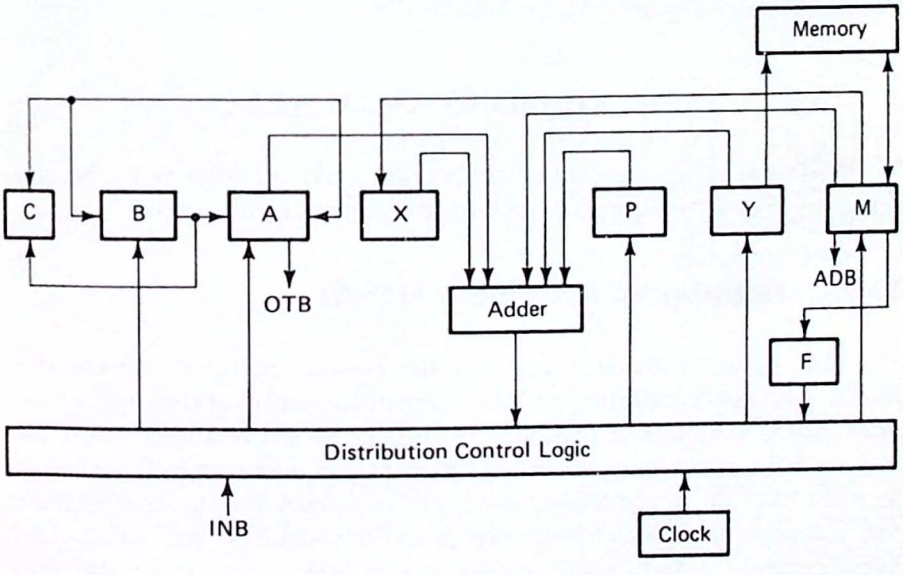


FIGURE 10-1. Simplified block diagram (H316 or DDP516).

M, Y, and D-registers) are at times divided into a high and low part to facilitate split word operations. This feature is used to implement five byte-manipulating instructions. The contents of the A-register are interchanged, or interchanged and either half cleared, or either half cleared by using the byte-manipulating instructions.

The op code decoder of the distribution control logic (DCL) contains binary-to-octal converters and control gates to generate enable signals for specific instructions. The F-register provides inputs to the binary-to-octal converters. Control gates utilize the converter outputs and two bits of the M-register to generate enable signals to generic and jump/skip instructions. The DCL contains all gates and control flip-flops necessary for proper instruction execution, including the C-bit flip-flop used to indicate overflow on arithmetic instructions. It is also used, in conjunction with a six-bit counter, in shift instructions.

The clock operates at a basic oscillator frequency of approximately 2 mHz. The 1.92 μsec memory cycle is divided into four 480 nanosec time periods. Control logic for shift instructions can cause the clock to repeat a specific number of times.

The I/O bus and control communicate with peripheral devices via PCUs to move data in and out of the computer. The bus arrangement for direct I/O is described in Sec. 5-10. During output, data transmission is from the A-register to the bus. During input, data transmission is from the bus to the A-register, but passing through the D-register first.

The memory is coincident-core provided in optional (after the first 4K) modules. Maximum system memory is 32K and is determined by the number of 4K modules in the system. A full memory cycle (read and write) requires 1.92 μsec. Access time is 0.96 μsec.

10-4 OPTIONS AVAILABLE

With the H316 one may select one or more options with the basic computer. Some of the major options are summarized below.

10-4.1 High-Speed Arithmetic (HSA)

This option provides logic for hardware multiply, divide, and double-precision transfers. The basic computer without this option provides multiplication and division by means of subroutines that effectively calls for successive additions for multiply and successive subtractions for division. In addition, the HSA provides for double-precision data transfers, as well as for double-precision addition and subtraction. Approximately a ten-to-one speed advantage exists over the pure software method for multiply using the HSA option.

10-4.2 Real-Time Clock

This option readily provides the user with program-controlled timing delays within the system. A modest amount of logic adds several instructions that assist in the control of real-time systems.

10-4.3 Additional Memory

Anywhere from one to eight memory modules may be supplied with a H316. For some programs, for example, FORTRAN, at least 8K of core is required. With additional memory, programs and subroutines can be continuously stored in core, thus, perhaps, avoiding external peripheral storage, its expense, and possible slowing of operations.

10-4.4 Direct Multiplex Control (DMC)

This is an abridged form of DMA. Control of external data movements to or from the processor memory is supervised by the peripheral. This option permits up to 20 devices to be connected to the computer by a separate interface. The memory address for each DMC transfer is stored in a specific memory location. This is incremented at each transfer time; thus the address stays current. A call for a DMC transfer halts the on-going computer program long enough to effect the transfer. DMC interrupts the on-going program at the end of an instruction execution. Although DMC substitutes memory storage of transfer addresses as contrasted with DMA's hardware registers, it is a less costly form of DMA.

10-4.5 Peripheral Devices

A teletype keyboard-printer with paper tape punch and reader is the minimum peripheral that one can live with; many others are available on order. These include card readers, high-speed paper tape readers and punches, high-speed line printers, magnetic tapes, disk files, and communication modems.

10-4.6 Priority Interrupts

A multilevel interrupt system is available and eliminates the necessity to query the single standard interrupt line with SKS instructions until the interrupting device is found. This option is furnished in groups of four with a separate line for each interrupt. A unique memory address

is dedicated to each interrupt line. These addresses contain a unique jump instruction that points to a subroutine to service the interrupt. Included with this option is a register settable by computer program to mask out (delete) any of the interrupts to which the operator does not want the computer to respond. Up to 12 groups of four-line priority interrutps may be provided, a total of 48 altogether.

10-5 USING THE ASSEMBLER

Even though one could insert one program instruction or data word at a time via control panel switches, this is an unnecessarily tedious process and is hardly ever followed except for very simple programs and testing. Programs are usually written in *assembly* or *compiler language form, and the source programs* are then converted by a *DAP-16 assembler* or a compiler (usually FORTRAN) to machine code. Often the source program is inserted via paper tape or via Hollerith cards, with the assembler (compiler) already in core, possibly loaded previously from disk or tape storage. The machine language resulting from the combined action of source program and assembler (compiler) may be restored on the external peripheral file until needed. If it is reasonably short then a punched paper tape or punched cards provide a suitable storage form.

The H316 Programmers Reference Manual lists the commands that are to be used as well as macroinstructions (pseudo-operations) available the DAP-16 assembler. The sequence of instructions and data in the source program to be assembled are examined twice by the assembler, once to develop a dictionary of symbols and a second time to assemble the *object program* by referencing the dictionary. Printing of each assembled instruction during pass number 2 is completed before the next line is started. The source program line is read, stored in a special buffer (memory), the data word or instruction assembly is performed and, if requested, the line is printed and an object program is tape punched, or recorded. Figure 10-2 illustrates symbolically how each line is processed. The assembler calls the subroutine for reading and storing one program line. The line is separated into its constituents (fields) and the operation is mnemonic examined. Whether the operation to be performed is *normal* or *pseudo* determines the subroutines called to process the field. For normal operations the assembler determines the machine operation by a table lookup and places the op code in the proper position of the assembled instruction. For pseudo-operations, analytical subroutines serve to modify the assembly process, allocate storage, define data words, or provide program linkages at load time. For more details, including a

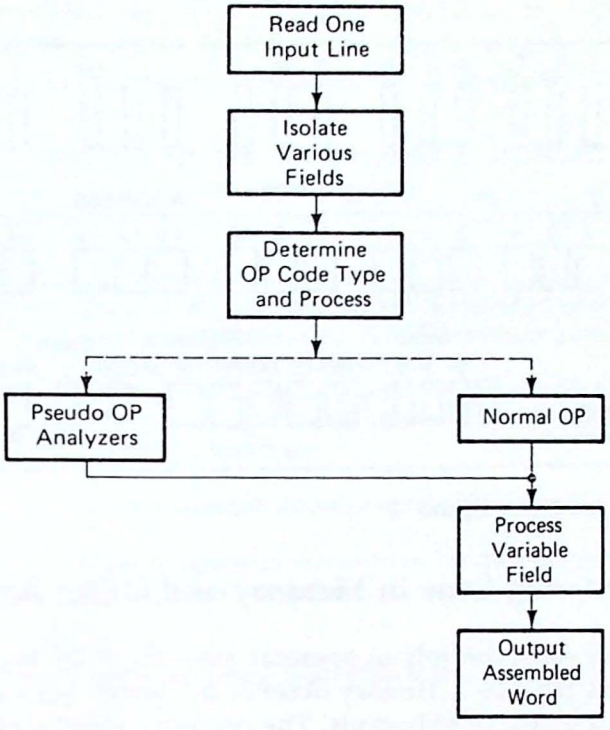


FIGURE 10-2. Processing of one line (from H316 Programmer's Reference Manual).

one-pass assembler, the reader is referenced to appropriate Honeywell documentation.

The pseudo-operations for DAP-16 are defined in Chapter 5 of the *Programmer's Reference Manual,* along with eight assembler format blocks. The DAP-16 assembler is most valuable for control-type programs. A FORTRAN compiler is likewise available. In the manual's appendix there is listed a total of 137 mathematical, 18 test, 65 I/O, and 15 other utility subroutines. Each user receives via a paper tape those subroutines usable with his particular system.

10-6 CONTROL PANEL SWITCHES

10-6.1 Control Panel Switches Listing

Table 10-1 lists the individual functions of the control panel switches and displays. Normally the teletype is in its operational mode. Figure 10-3 pictures the controls and displays of the H316 control panel.

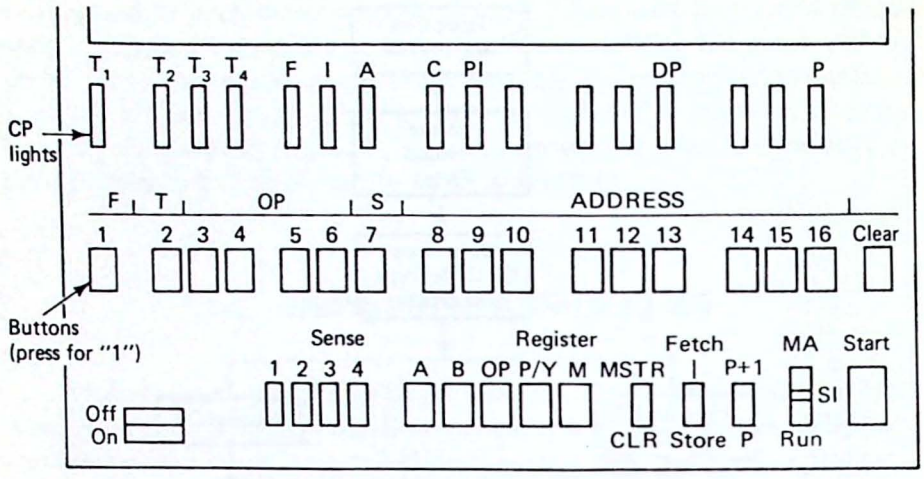


FIGURE 10-3. H316 control panel.

10-6.2 Entering Code in Memory and Other Registers

Via front panel controls an operator may select the registers or displays listed in Table 10-1. He may observe or change the data by selecting the A, B, OP, PY, or M buttons. The present contents of the selected register are readable in octal (or binary) via the 16-bit lights of the front panel. If he wishes to change the contents, he may press the CLEAR button, clearing the present contents, and then press those buttons underneath the lights where he wants a (1) in the new contents. Note that both data and commands may be inserted in this way. The reader should reference the H316 *Operator's Guide* for complete instructions. The 16-bit display is illustrated by Fig. 10-4, showing both octal and binary equivelents. It is common in Honeywell literature to express each 16-bit word in octal rather than binary.

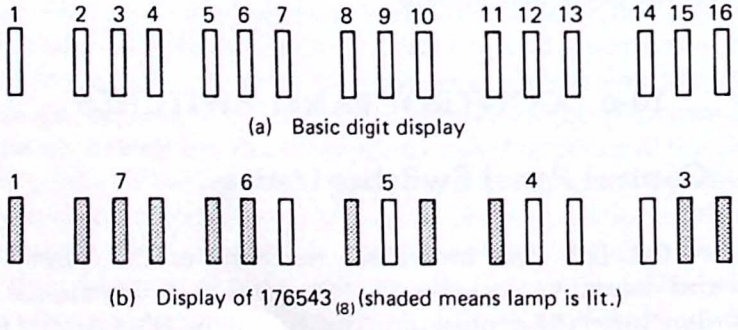


FIGURE 10-4. Control panel 16-bit displays.

For example, in Fig. 10-4(b) $176543_{(8)}$ is the same as $1111110101100011_{(2)}$. Table 10-1 lists the functions of the H316 control panel switches. For computer operation the TTY must also be turned on.

TABLE 10-1 Functions of the control panel pushbuttons (See H316 operator's guide)

Pushbutton	*Function*
Indicators 1 through 16	Enter or display data. A ONE is indicated when pushbutton/indicator is illuminated. A ZERO is indicated when pushbutton/indicator is not illuminated. For the H316, data are entered by depressing switch 1 through 16; data are displayed by the lights immediately above the switches.
CLEAR	Clears selected display register (A, B, P/Y or M only).
POWER-ON (ON switch for 316)	Applies power to main frame. The indicator is illuminated when power is applied.
POWER-OFF (OFF switch for H316)	Removes power from main frame.
X	Displays contents of index register (516 only). (Data can be entered into register X by means of location 00000_8)
A	Displays and permits alterations in contents of register A.
B	Displays and permits alterations in contents of register B.
OP	Displays state of key flip-flops in the system (see Table 1-4).
P/Y	Displays and permits alterations in contents of registers P and Y. [Registers P and Y contain the same address when the computer is in the memory access (MA) mode. Register P contains an address that is one higher than that contained in register Y when the computer is in the single instruction (SI) or run modes.]
M	Displays and permits alterations in contents of register M
MSTR CLEAR	Clears registers A, B, P, Y and M; sets all timing to that state existing following a halt; initializes standard peripheral devices and options
START	Starts machine in all modes. Indicator is illuminated when machine is in run mode.

10-7 USE IN CONTROL APPLICATIONS

The minicomputer is especially useful for control applications. For example, in the Minuteman III program at Honeywell each production test set had its own dedicated H316 for control of testing the board modules, the memory assembly, and the assembled computer for Minuteman III (H701). A few years previously the arsenal checkout computer was a single IBM 1800 that controlled by itself all test stations. If it failed, for any reason, the entire testing program stopped. Also, the programming for such a large centralized system is more difficult to debug. With a single computer at each individual test station a computer failure stops only one test, not the entire system of testing. H316s are priced so much less than the other computer that on expense alone they can be justified at each station.

10-8 INTERFACING PROBLEMS AND HOW TO AVOID THEM

The first rule of interfacing is to follow the manufacturer's interface manual to the letter. This is especially true of strobe timing, strobe widths, etc. It generally has proved best (especially if long lines are involved) to employ differential line drivers, twisted-pair wires, and differential line receivers at the receiving end. Typical examples of these are the National Semiconductor Corp. N8430 line receiver and N8420 line transmitter.

In the H316 the address bus comes directly from the M-register (MBR). The problem here is that all data in and out of memory appear at this register. It is possible to protect PCUs from spurious codes (not intended to be instructions to the PCU) by opening a path within the PCU by a flip-flop set with an OCP command. The next M-register contents will then be read as a command by the PCU involved. The path is then closed by resetting the flip-flop with a second OCP command; thus, no spurious codes can be interpreted by the PCU as a command.

10-9 MINIS COMPARED

Table 10-2 summarizes a representative group of minicomputers that have recently been or are now popular among users. It should be noted that in this fast-moving industry, no particular machine holds first place indefinitely. In the Honeywell line the H316 descends from the DDP516, which in turn descended from the earlier DDP116. Improve-

TABLE 10-2 Representative minicomputers

Manuft.	*Honeywell*	*Varian*	*Texas Inst.*	*Comp. Auto*	*Interdata*	*Data Gen.*
Model #	*H316*	*620i*	*TI960*	*Alpha LSI-305A*	*7/16*	*Nova 3/12*
Size (in.) (H × W × D)	17 × 20 × 30	12½ × 19 × 22	12½ × 19 × 24	9 × 19 × 18	7 × 19 × 19	11 × 19 × 23
Weight (lb)	—	70	75	—	50	130
Power (watts)	600	340	420	120	414–690	600
No. Instr.	72	100	76	95	108	377
I/O types	DIO DMC	DIO DMA	DIO DMA	DIO DMA	DIO DMA	DIO DMA
Interrupts	1 line options	8 line options	8-lines	Vectored priority	255 priority	16 priority
Memory	4–32K core	4–32K core	4–64K MOS	4–16K MOS	4–65Kbytes Core MOS	4–128K words MOS
Mem. cycle (μsec)	1.92	0.96	0.75	0.300	0.75	0.700
Add time (μsec)	3.94	3.6	3.6	6.4	1–3.75	1.1
Software	yes	yes	yes	yes	yes	yes
Word bits	16	16(i7)	16	16	16/32	16
Hardware multiply	Opt.	Opt.	—	—	Opt.	Opt.
	First-generation machines			Third- or four-generation machines		

Note: "—" means that when table was prepared data were not available from data sheets. Most of these machines have been superseded by new models, for example, the Honeywell H716 and Data General's MicroNOVA, etc.

ments in packaging and components justified the design of a new model. Although the H316 is a fraction of the physical size of the DDP516, and less costly, it has only half the speed. Now both machines have been superceded by the model H716. Software features of the original DDP116 have been passed on and improved in succeeding models. The H716 operates in many ways as its predecessors but has added features unique to itself, such as:

1. DMA is included and not an option.
2. RO and RI registers control the DMA. (Note: the DDP516 had a DMA, but only as an option. The H316 approximated a DMA by its DMC channels.)
3. The I/O buses are all bidirectional, thus saving wires. The Address bus, being bidirectional, allows addresses from PCUs to be directed inward directly to memory, making all functions of a DMA channel outside of the computer. Direct I/O transfers and DMA data transfers occur over the same set of wires (not separate sets, as in the DDP516 DMA). The bidirectional data bus is identified as PIO.
4. A special plug-in card permits H316 peripherals to be operated directly by the H716. This card regenerates the standard H316 separate data-in and data-out buses along with the standard address bus and timing strobes.
5. The real-time clock is now standard—not an option.
6. Programs written for earlier computers of the line will operate on the H716.
7. There are a few new instructions executable on the H716 alone. Some of these control the H316 interfacing board of item 4 above.

10-10 SUMMARY

This chapter has attempted to summarize the explosive field of minicomputers. The H316, although outmoded by newer machines, is employed as a representative example, primarily because the author has employed this machine in automated testing applications. Admittedly, many of the newer machines have advantages over the H316, but this computer builds on previous software, and employs many proved peripherals in its product line. It is easy to interface if the precautions of Sec. 10-8 are heeded. The INTERDATA Computer discussed in Chapter 11 is hexadecimal rather than octal as the H316 is.

10-11 PROBLEMS

1. What are some of the characteristics of a minicomputer?
2. What software is usually supplied by the manufacturer?
3. What are the functions provided in the diagram Fig. 10-1 (explain the registers, buffers, buses, etc.)?
4. What are some of the options available to the H316?
5. Explain Fig. 10-2.
6. What are the functions of the A, B, OP, and M buttons?
7. How are ASC11 codes displayed on the H316?
8. Write -57213 in binary code on the display.
9. How do you avoid interfacing problems in a specially built PCU? What is DMC a substitute for in other computers?
10. Write in bit form ASC11 codes for S and 2 in the display with odd parity, shown as a (1).

11

The Interdata 7/16 Computer

11-1 SYSTEM ARCHITECTURE (CPU)

11-1.1 General

The INTERDATA 16-bit processor is a third-generation hexadecimal (HD) number computer. The multicomputer architecture provides 16 general registers for maximum flexibility, and thus it requires less housekeeping tedium than machines without this advanced feature. None of the 16 registers is dedicated to any specific use such as index registers, stack pointers, program counter, or subroutine return pointers. The programmer is free to use these for partial storage, of such items as number constants, loop management constants, etc. The central processing unit (CPU) has a program status word register (PSW), the general register mentioned above, signed multiply and divide hardware (option), and the floating point (FP) hardware (option), including eight single precision (32-bit) registers.

11-1.2 Memory

The main memory is usually core modules available in 8KB, 16KB, 32KB, and 64KB modules (KB is *kilobytes*). The modules are available with parity to increase functional reliability. Most modules have a 1 μsec cycle time, but in the two latest versions it is possible to have a 750 nsec cycle time.

11-1.3 Standard Software

Includes:

Symbolic assembler	FORTRAN IV
Basic operating system	BASIC

Interactive test editor
Instructive debug package

11-1.4 Instruction Set

The processor has 90 basic instructions. Instructions use both 16-bit and 32-bit formats. Operations are between general registers and memory. The 64-bit precision floating point instructions are not discussed because our equipment did not have this option. Reference should be made to the *16-bit Processors User's Manual* for a complete programmer's description of each instruction. Instructions provide a complete set of logical operations, control operations, arithmetic operations, and input/output operations on a byte or halfword basis. A byte is 8 bits, and a halfword is 16 bits.

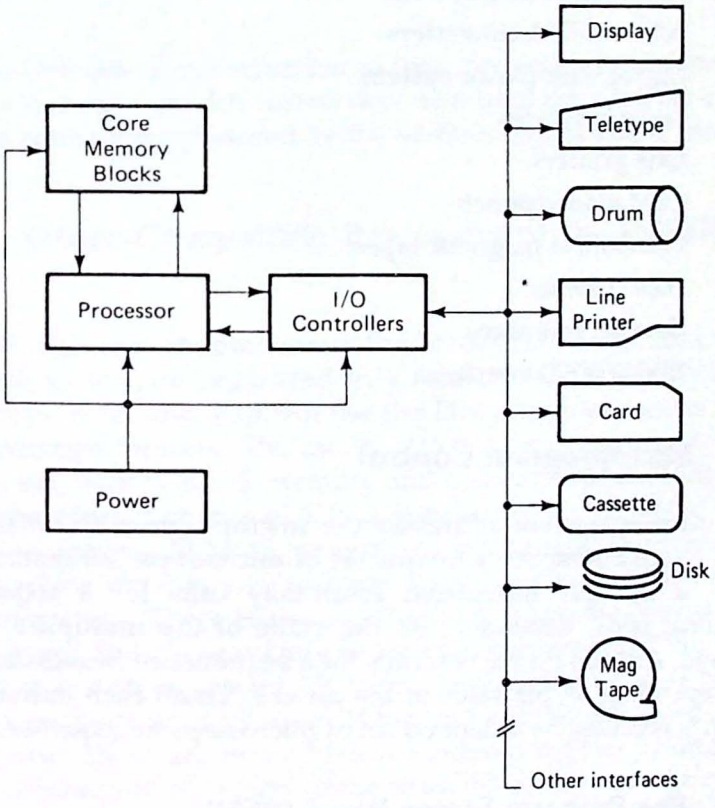


FIGURE 11-1. INTERDATA Model 7/16 simplified system block diagram.

11-1.5 Input/output

There is a dual input/output bus structure handling up to 255 devices including the display panel. High-speed devices via the direct memory access channel (DMA) operate up to 2,000,000 bytes per second; medium- to slow-speed devices, up to 60,000 bytes per second. Available are both program direct control and interrupt automatic vectoring via an interrupt service table.

11-1.6 Peripheral Products Available

1. Selector channel
2. Teletype
3. Cathode ray tube (CRT)
4. Tape cassette
5. Real-time analog system
6. A/D and D/A converters
7. Digital multiplexor system
8. Logic interfaces
9. Line printers
10. Card reader/punch
11. Compatible magnetic tapes
12. Disk systems
13. Data set interfaces
14. IBM 360/370 interfaces

11-1.7 Microprogram Control

A macroinstruction addresses the microprogram ROM memory, where for each instruction a unique set of microsteps is initiated. As an example, a multiply instruction essentially calls for a sequence of hexadecimal adds, depending on the value of the multiplier. On the other hand, a divide instruction calls for a sequence of hexadecimal subtracts, depending on the value of the divisor. Thus, each individual instruction is executed by a defined set of microsteps in sequence.

11-1.8 The Program Status Word (PSW)

The PSW defines the state of the processor at any given time. It is a 32-bit word, and the functions of the various bits of the word are de-

scribed in Sec. 11-7.1 and 11-7.2. In Sec. 11-7.2 Table 11-7 defines the various instructions relating to the PSW. The 11 MSBs indicate the processor status, bits 12–15 are the condition code, and bits 16–31 are the instruction location counter. Bits 12–15, the condition code, are set during most instructions during their execution and are used to signal the instruction's function to the remainder of the processor's logic. For example, during a typical instructions execution the condition code bits may be set according to the following tabulation.

Bits	*12*	*13*	*14*	*15*	
Code	*C*	*V*	*G*	*L*	
	0	0	0	0	Equal comparison or results = 0
	0	0	0	1	Less than
	0	0	1	0	Greater than
	0	1	0	0	Overflow
	1	0	0	0	Carry or Borrow

Each instruction will have its own table. See INTERDATA 16-Bit Processor User's Manual.

The LSB bits, the instruction counter, are normally incremented by 1, except during a branch instruction, and then the contents of the instruction counter are replaced by the address as to where to branch.

11-1.9 Other Compatible Processors of the INTERDATA Family

Although this chapter treats the Model 7/16 only, other INTERDATA processors are organized in a similar manner and include the Modles 5/16, 6/16, and 8/16. All use the IBM computer models 360/370-type instruction formats. The model 5/16 is a small one-board machine having a 600 nsec N-MOS memory and a microprocessor bidirectional input/output bus. The model 6/16 is very similar to the model 7/16 but offers as an option a N-MOS memory in place of core. The model 8/16 is a core machine but offers in addition several double-precision floating-point instructions. There is also a 7/32 model, which is a complete 32-bit minicomputer and consequently its general registers are 32 bits long. In addition, it has an expanded RX format, containing up to 47 bits. This allows an absolute address up to 24 bits and a 32-bit data word in the RI format. Also, there are two additional indexed register 32-bit formats. These machines are all typical of the latest minicomputer organizations, which are powerful enough to perform business processing for a long time restricted to the large computers—hardly a function for the first-generation process control minicomputers.

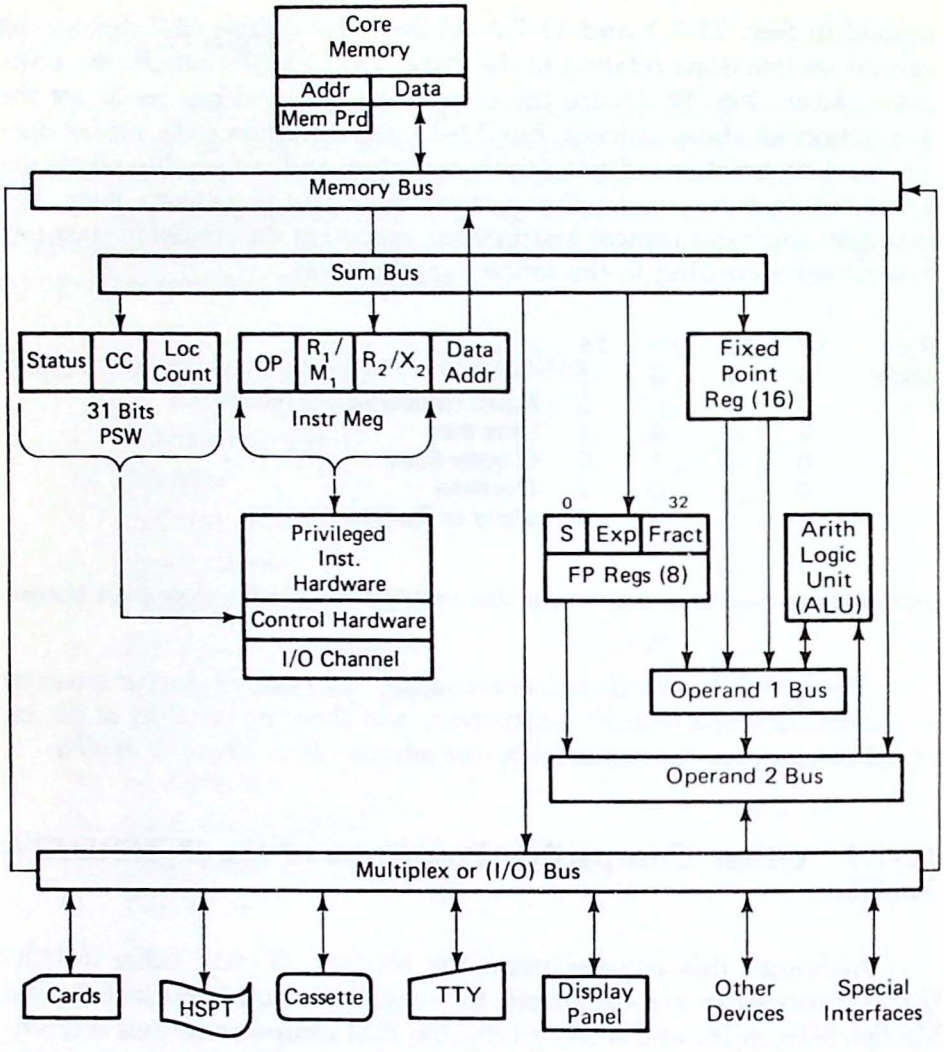


FIGURE 11-2. Processor block diagram (Interdata models 70, 74, and 80).

11-2 THE HEXADECIMAL (HD) DISPLAY PANEL

This panel provides a manual means to control the processor, to integrate various machine states (such as the PSW), the program steps, and contents of memory locations and registers, and may be programmed to act as an I/O device (see *Display HD bytes* in the test program section 11-9.12). It is equipped with both binary and hexadecimal (HD) readouts, as illustrated in Fig. 11-3. Means are provided to enter HD data, memory addresses, and commands. Also, these entries can be accessed at both register and memory level. Among other items are control keys that read the entered program (RD), enter the address (ADD), enter

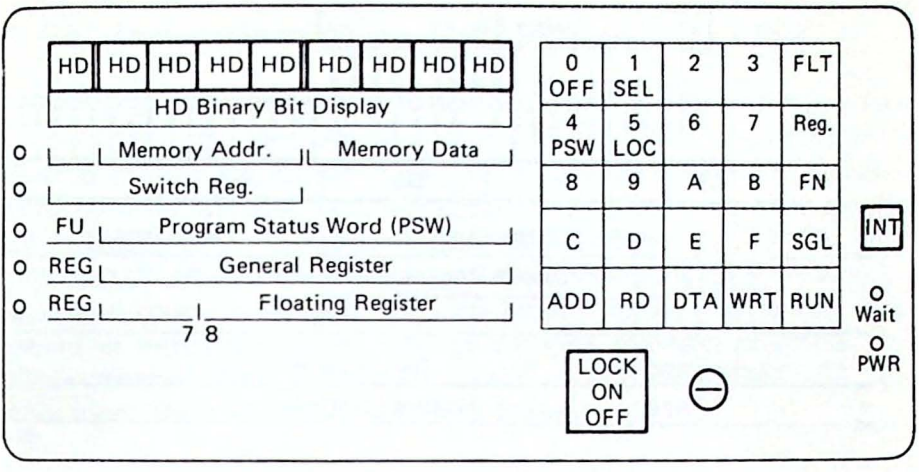


FIGURE 11-3. Hexadecimal display panel.

data or commands by clearing the switch register (DTA), and single-step the program (SGL), a key to interpret special functions (FN), a key to examine general registers (REG), a key to examine FP registers (FLT), an off/on key, a key to write into memory (WRT), and a (RUN) key. *The 16-bit Processor Manual* contains details on how to operate this display/control panel and should be consulted for more detail.

11-3 INSTRUCTION FORMATS

11-3.1 General

There are four basic instruction formats employed for normal operation of the computer (see Fig. 11-5): R1, first operand address; R2 and A, second operand addresses; N, a 4-bit immediate value; B, an immediate value; X2 equals index register specified. X2 is added to A and B.

11-3.2 Format Details

RR R1 equals first operand; R2 equals second operand

SF R1 equals first operand; N is a field of four bits immediate value. N can be + or −.

RX R1 equals the first operand; X2 contains indexed quantity; A is the second operand address.

RI R1 equals the first operand; X2 contains indexed quantity; B is the immediate value.

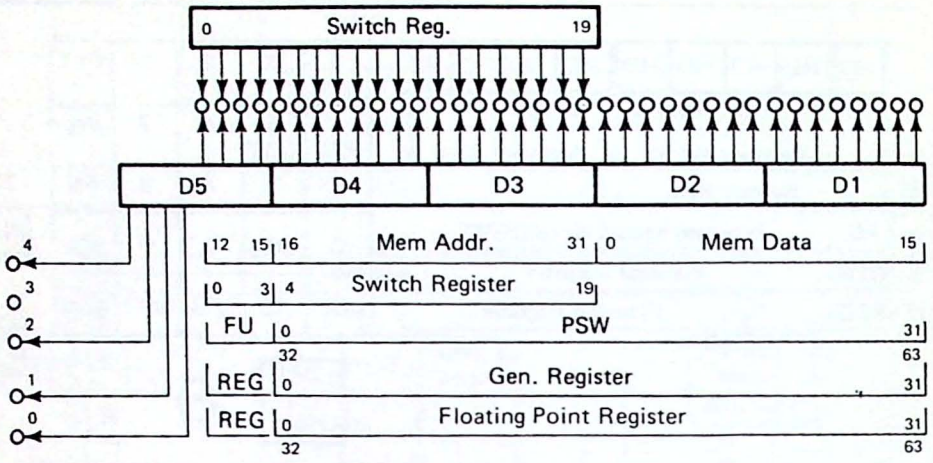
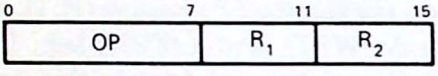
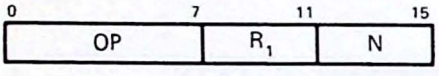


FIGURE 11-4. Display registers and indicators.

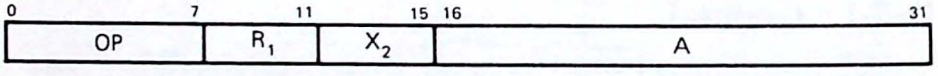
Register to Register (RR)



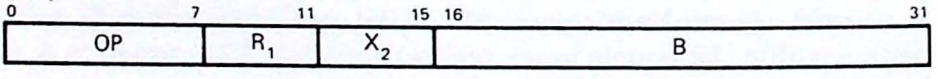
Short Format (SF)



Register and Indexed Storage (RX)



Register and Immediate Storage (RI)



OP OP code
R_1 First operand register
R_2 Second operand register
N A 4-bit immediate value
X_2 Second operand index register
A Second operand address
B Second immediate data

FIGURE 11-5. Instruction formats.

11-3.3 Instructions of the Data Manipulating Class

Instructions in this class include the load, store, and byte instructions listed in Table 11-1. These instructions control loading of data, storing data, and the manipulation of 8-bit bytes of data. Table 11-2 includes instructions for shifting data to the left or right, rotation of data, etc. Finally, Table 11-3 includes logical instructions—i.e., AND, OR, EXCLUSIVE-OR and data words. All the instructions listed are in tabular form in order to simplify the detail that a reader must absorb. This amount of detail should be enough to write complete programs. For more extensive detail the reader should consult the INTERDATA Inc. publication: *The 16-bit Processor User's Manual.*

11-4 BRANCHING

11-4.1 General

Normally the processor executes instructions in sequential order. The branch instructions permit a variation of this order so that the program can loop, transfer control, or make decisions based on previous conditions. In many of these, a four-bit mask is compared with the condition code. This provides either a jump to the second operand address or continuing on to the next instruction. Included in this group of instructions are those that branch to subroutines and return to the main program.

11-4.2 Branching Instructions

These instructions occur in the RX, RR, and SF formats. The RX and RR formats are employed in the past section. The SF format in this section specifies a jump forward or backward an N number of halfwords in the program. The majority of branching instructions are conditional branches based upon the setting of the condition code in the PSW word. For one condition code the instruction will cause a branch to the second operand location. For the other condition code setting of interest, if it occurs, the next sequential instruction is executed—i.e., the branch does not occur. All the branching instructions are listed in Table 11-4. The list concludes with three unconditional branch instructions (B, BR, and BS) and the two no-op instructions NOR and NOPR. For the exact condition code settings involved for each instruction the reader should refer to the *16-bit Processor User's Manual.*

TABLE 11-1 Load, store, and byte instructions

Instruction	*Mnemonic*	*Operand*	*Op-code*	*Format*	*Explanation*
Load Halfword	LH	R1, A(X2)	48	RX	Load the second operand contents into R1.
Load Halfword Register	LHR	R1, R2	08	RR	
Load Halfword Immediate	LHI	R1, B(X2)	C8	RI	
Load Immediate Short	LIS	R1, N	24	SF	Load N into R1.
Load Complement Short	LCS	R1, N	25	SF	Load $\bar{N}$ into R1.
Load Multiple	LM	R1, A(X2)	D1	RX	Starting with R1 successive registers are loaded starting at A(X2). When R15 loaded process stops.
Store Halfword	STH	R1, A(X2)	40	RX	R1 is loaded in second operand address.
Store Multiple	STM	R1, A(X2)	DO	RX	Starting with R1 successive memory locations are loaded in sequence. When R15 loaded process stops.
Load Byte	LB	R1, A(X2)	D3	RX	The second operand replaces bits 8:15 of R1. Bits 0:7 are forced to zero.
Load Byte Register	LBR	R1, R2	93	RR	
Store Byte	SB	R1, A(X2)	D2	RX	Bits 8:15 of R1 are stored in second operand location. Two eight-bit bytes of R2 are exchanged and put in R1.
Store Byte Reg.	SBR	R1, R1	92	RR	
Exchange Byte	EXBR	R1, R2	94	RR	
Compare Logical Byte	CLB	R1, A(X2)	D4	RX	Bits 8/15 of R1 are compared with second operand. The result sets up the condition code.

TABLE 11-2 Shift and rotate instructions

Instruction	*Mnemonic*	*Operands*	*Op-code*	*Format*	*Explanation*
Shift Right Halfword Log.	SRHL	R1, B(X2)	CC	RI	These instructions shift R1 right the number of places specified by the second operand contents. For the fullword instructions R1 contains the MSBs and R1 + 1 contains the LSBs. Bits shifted right out of R1 go to R1 + 1. Bits shifted right out of R1 + 1 are lost. For the arithmetic instructions the sign bit is not shifted
Shift Right Halfword Arith.	SRHA	R1, B(X2)	CE	RI	
Shift Right Log. Fullword	SRL	R1, B(X2)	EC	RI	
Shift Right Arith. Fullw'd	SRA	R1, B(X2)	EE	RI	
Shift Left Halfword Log.	SLHL	R1, B(X2)	CD	RI	These instructions shift R1 + 1 left the number of places specified by the second operand contents. Bits shifted from R1 + 1 go left into R1. Bits shifted out left in R1 are lost. For the fullword instructions MSBs are in R1 and LSBs in R1 + 1. For the arithmetic instructions the sign bit is not shifted.
Shift Left Halfword Arith.	SLHA	R1, B(X2)	CF	RI	
Shift Left Log. Fullword	SLL	R1, B(X2)	ED	RI	
Shift Left Arith. Fullword	SLA	R1, B(X2)	EF	RI	
Shift Left Log. Short	SLLS	R1, N	91	SF	Load N into R1 for SF.
Shift Right Log. Short	SRLS	R1, N	90	SF	Load N into R1 for SF.

(Cont.)

TABLE 11-2 (Cont.)

Instruction	*Mnemonic*	*Operands*	*Op-code*	*Format*	*Explanation*
Rotate Right Log.	RRL	R1, B(X2)	EA	RI	Registers R1 and R1 + 1 are linked together. For RRL the contents of R1 shift into R1 + 1 and the contents of R1 + 1 shift out and back into R1 left side. For RLL the opposite holds true. Bits shifted out left of R1 are shifted back into R1 + 1 right-hand side.
Rotate Left Logical	RLL	R1, B(X2)	EB	RI	
Add to Top of List	ATL	R1, A(X2)	64	RX	R1 contains the halfword elements to be added to the list in memory at the second operand address as manipulated by the top of the list pointer or next to the bottom of the list pointer.
Add to Bottom of List	ABL	R1, A(X2)	65	RX	
Remove from Top of List	RTL	R1, A(X2)	66	RX	The list is located at the second operand location. The element halfword removed from the list replaces the contents of R1.
Remove from Bottom of List	RBL	R1, A(X2)	67	RX	

Log. is the abbreviation for logical.

TABLE 11-3 Logical instructions

Instruction	*Mnemonic*	*Operand*	*Op code*	*Format*	*Explanation*
AND Halfword	NH	R1, A(X2)	44	RX	The logical AND of R1 and the second operand replaces R1 contents.
AND Halfword Register	NHR	R1, R2	04	RR	
AND Halfword Immediate	NHI	R1, B(X2)	C1	RI	
Test Halfword Immediate	THI	R1, A(X2)	C3	RI	The logical AND of R1 and the second operand. Neither operand is changed.
OR Halfword	OH	R1, A(X2)	46	RX	The logical OR of R1 and the second operand replaces R1 contents.
OR Halfword Register	OHR	R1, R2	06	RR	
OR Halfword Immediate	OHI	R1, B(X2)	C6	RI	
Exlcusive OR Halfword	XH	R1, A(X2)	47	RX	The logical Exclusive OR of R1 and the second operand replaces R1 contents.
Exclusive OR Halfword Reg.	XHR	R1, R2	07	RR	
Exlcusive OR Halfword Imm.	XHI	R1, B(X2)	C7	RI	
Comp. Logical Halfword	CLH	R1, A(X2)	45	RX	R1 is compared to the second operand which results in a unique condition code setting.
Comp. Logical Halfword Reg.	CLHR	R1, R2	05	RR	
Comp. Logical Halfword Imm.	CLHI	R1, B(X2)	C5	RI	

TABLE 11-4 Branching instructions

Instruction	*Mnemonic*	*Operand*	*Op code*	*Format*	*Explanation*
Branch on True Condition	BTC	M1, A(X2)	42	RX	The condition code of the PSW is tested per mask field, M1. If any conditions are TRUE, a branch is executed to the second operand location. If none is TRUE, the next sequential instruction is executed.
Branch on True Condition Reg.	BTCR	M1, R2	02	RR	
Branch on True Condition Fwd.	BTFS	M1, N	21	SF	Load N into R1 for SF.
Branch on True Condition Bwd.	BTBS	M1, N	20	SF	Load N into R1 for SF.
Branch on False Condition	BFC	M1, A(X2)	43	RX	The condition code of the PSW is tested per mask field, M1. If all conditions FALSE, a branch is executed to the second operand location. If any TRUE, the next sequential instruction is executed.
Branch on False Condition Reg.	BFCR	M1, R2	03	RR	
Branch and Link	BAL	R1, A(X2)	41	RX	The address of the next sequential instruction is saved in R1, and a branch is taken to the second operand location.
Branch and Link Register	BALR	R1, R2	01	RR	
Branch on False Condition Fwd.	BFFS	M1, N	23	SF	Load N into R1 for SF.
Branch on False Condition Bwd.	BFBS	M1, N	22	SF	Load N into R1 for SF.
Branch on Index Low or Equal	BXLE	R1, A(X2)	C1	RX	Prior to execution R1 contains the starting index value, R1 + 1, the increment value, and R1 + 2, the final value. The increment value is added to the index value. The result is compared to the final value. If index is less than final value, a branch is executed to the second operand location. If index value is greater than final value, the next sequential instruction is executed.

Branch on Index High	BXH	R1, A(X2)	CO	RX	R1, R1 + 1, and R1 + 2 contain the same constituents as for BXLE, except R1 is the upper-end register, R1 is minus, and R1 + 2 cannot be less than RO. As before, the increment value is added to the index value, and a comparison is made. If the index value is greater than the final value, a branch is made. Otherwise, the next sequential instruction is executed.
Branch (uncond.)	B	A(X2)	430	RX	A branch is executed to the second operand address unconditionally.
Branch (uncond.) Register	BR	R1, R2	030	RR	
Branch (uncond.) Short	BS	R1, N	230 Fwd. 220 Bwd.	SF	Load N into R1.
No operation	NOP	A(X2)	420	RX	After a short delay the next instruction is executed. A(X2) and R2 are usually equal to 0.
No operation Register	NOPR	R1, R2	020	RR	
Branch on Carry	BC	R1, A(X2)	428	RX	If the C flag of the condition code is set, a branch occurs to the second operand address. If C flag is reset, the next sequential instruction is executed
Branch on Carry Register	BCR	R1, R2	228	RR	
Branch on Carry Short	BCS	R1, N	218 Fwd. 208 Bwd.	SF	Load N into R1 for SF

(Cont.)

TABLE 11-4 (Cont.)

Instruction	*Mnemonic*	*Operands*	*Op-code*	*Format*	*Explanation*
Branch on no Carry	BNC	R1, A(X2)	438	RX	If C flag of condition code is reset, a branch occurs to the second operand address. If the C flag is reset, the next sequential instruction is executed.
Branch on no Carry Reg.	BNCR	R1, R2	038	RR	
Branch on no Carry Short	BNCS	R1, N	238 Fwd. 228 Bwd.	SF	Load N into R1.
Branch on Equal	BE	R1, A(X2)	433	RR	If the G and L flags of the condition code are both reset, a brance is executed to the second operand address. If either flag is set, the next sequential instruction is executed.
Branch on Equal Register	BER	R1, R2	033	RR	
Branch on Equal Short	BES	R1, N	223 Fwd. 233 Bwd.	SF	Load N into R1.
Branch on not Equal	BNE	R1, A(X2)	423	RX	If the G or L flag of the condition code is set, a branch is executed to the second operand address. If neither flag is set, the next sequential instruction is executed.
Branch on not Equal Reg.	BNER	R1, R2	023	RR	
Branch on not Equal Short	BNES	R1, N	213 Fwd. 203 Bwd.	SF	Load N into R1.

Branch on Low	BL	R1, A(X2)	428	RX	When the two operands are compared, the PSW condition code C flag is set if the first operand is less than the second operand. If the C flag is set, a branch is executed to the second operand address. If the C flag is reset, the next sequential instruction is executed.
Branch on Low Register	BLR	R1, R2	028	RR	
Branch on Low Short	BLS	R1, N	218 Fwd. 208Bwd.	SF	
Branch on not Low	BNL	R1, A(X2)	438	RX	When the two operands are compared, the C flag of the condition code is reset if the first operand is not less than the second operand. If the C flag is reset, a branch is executed to the second operand address. If the C flag is set, the next sequential instruction is executed.
Branch on not Low Register	BNLR	R1,R2	038	RR	
Branch on not Low Short	BNLS	R1, N	238 Fwd. 228 Bwd.	SF	
Branch on Minus	BM	R1, A(X2)	421	RX	
Branch on Minus Register	BMR	R1, R2	021	RR	
Branch on Minus Short	BMS	R1, N	211 Fwd. 201 Bwd.	SF	Load N into R1 for SF.
Branch on not Minus	BNM	R1, A(X2)	431	RX	If the L flag of the condition code is set, a branch to the second operand address is executed. If the L flag is reset, the next sequential instruction is executed.
Branch on not Minus Reg.	BNMR	R1, R2	031	RR	If the L flag of the condition code is reset, a branch is executed to the second operand address. If the L flag is reset, the next sequential instruction is executed.
Branch on not Minus Short	BNMS	R1, N	231 Fwd. 221 Bwd.	SF	Load N into R1 for SF.

(Cont.)

TABLE 11-4 (Cont.)

Instruction	*Mnemonic*	*Operands*	*Op-code*	*Format*	*Explanation*
Branch on Plus	BP	R1, A(X2)	422	RX	If the G flag of the condition code is set, a branch to the second operand address is executed. If the G flag is reset, the next sequential instruction is executed.
Branch on Plus Register	BPR	R1, R2	022	RR	
Branch on Plus Short	BPS	R1, N	12 Fwd. 202 Bwd.	SF	Load N into R1 for SF.
Branch on not Plus	BNP	R1, A(X2)	432	RX	If the G flag of the condition is reset, a branch is executed to the second operand address. If the G flag is set, the next sequential instruction is executed.
Branch on not Plus Register	BNPR	R1, R2	032	RR	
Branch on not Plus Short	BNPS	R1, N	232 Fwd. 222 Bwd.	SF	Load N into R1 for SF.
Branch on Over flow	BO	R1, A(X2)	424	RX	If the V flag of the condition code is set, a branch to the second operand address is executed. If the V flag is reset, the next sequential instruction is executed.
Branch on Over-flow Register	BOR	R1, R2	024	RR	
Branch on Over-flow Short	BOS	R1, N	214 Fwd. 204 Bwd.	SF	Load N into R1 for SF.

Branch on no Overflow	BNO	R1, A(X2)	434	RX	If the V flag of the condition code is reset, a branch to the second operand address is executed. If the V flag is set, the next sequential instruction is executed.
Branch on no Overflow Reg.	BNOR	R1,R2	034	RR	
Branch on no Overflow Short	BNOS	R1, N	234 Fwd. 224 Bwd.	SF	Load N into R1 for SF
Branch on Zero	BZ	R1, A(X2)	433	RX	Branch on Zero If the G and L flags of the condition code are both reset a branch is executed to the second operand address. If the G or L flags are set, the next sequential instruction is executed.
Branch on Zero Register	BZR	R1, R2	033	RR	
Branch on Zero Short	BZS	R1, N	233 Fwd. 223 Bwd.	SF	Load N into R1 for SF.
Branch on not Zero	BNZ	A(X2)	423	RX	If the G or L flag of the condition code is set, a branch is executed to the second operand address. If the G and L flags are both reset, the next sequential instruction is executed.
Branch on not Zero Register	BNZR	R2	023	RR	
Branch on not Zero Short	BNZS	R1, N	213 Fwd. 203 Bwd.	SF	Load N into R1 for SF.

11-5 FIXED-POINT ARITHMETIC

11-5.1 General

Fixed-point arithmetic instructions provide a complete set to calculate addresses, indexing, and fixed-point arithmetic. For the INTERDATA computer the following table illustrates the range of values handled by the machines.

Hardware Format Relations

	HD	*Decimal*
Most negative	8000	-32,768
Least negative	FFFF	-1
Zero	0000	0
Least positive	0001	1
Most positive	7FFF	32,768

11-5.2 Arithmetic Instuctions

These instructions are the usual arithmetic operations of addition, subtraction, multiplication, and division. Also included are some additional instructions including additions with carry, subtractions with borrow, and several comparison instructions, similar to the logical comparison instructions listed previously in Table 11-2, but in this case these are arithmetic comparisons and consider the sign. For multiplication a 16-bit halfword multiply will result in a 32-bit product stored in the R1 and R1 + 1 registers with the LSBs in R1 + 1. The division of a 32-bit full word results in a 16-bit dividend and also a 16-bit remainder. These are stored as indicated in Table 11-5 (R1 + 1 containing the dividend and R1 the remainder). Table 11-5 describes all the fixed-point arithmetic instructions.

11-6 FLOATING-POINT ARITHMETIC

11-6.1 General

In base 10 one can write $186 = 1.86 \times 10^2$, or in scientific notation 186 also equals 0.186×10^3. Floating-point (FP) notation expresses numbers like 1.23×10^{23} much more compactly. But this computer is hexadecimal (HD), and conversion of numbers has to be to HD. In all FP

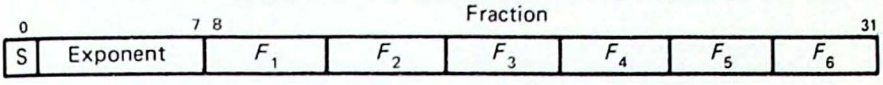
TABLE 11-5 Fixed-point Arithmetic instructions

Instruction	*Mnemonic*	*Operand*	*Op code*	*Format*	*Explanation*
Add Halfword	AH	R1, A(X2)	4A	RX	The second operand contents are added to R1 contents. The result is placed in R1.
Add Halfword Register	AHR	R1, R2	OA	RR	
Add Halfword Immediate	AHI	R1 B(X2)	CA	RI	
Add Immediate Short	AIS	R1, N	26	SF	
Add Halfword to Memory	AHM	R1, A(X2)	61	RX	The second operand contents are added algebraically to to R1 contents. The 16-bit result replaces the second operand contents.
Add with Carry Halfword	ACH	R1, A(X2)	4E	RX	The 16-bit second operand and the previous carry is added to R1 contents. The result is reflected in the condition code and replaces the contents of R1.
Add with Carry Halfword Reg.	ACHR	R1, R2	OE	RR	
Subtract Halfword	SH	R1, A(X2)	4B	RX	The second operand is subtracted from R1, and the result replaces the R1 contents.
Subtract Halfword Reg.	SHR	R1, R2	OB	RR	
Subtract Halfword Immediate	SHI	R1, B(X2)	CB	RI	
Subtract Immediate Short	SIS	R1, N	27	SF	Load N into R1.

(Cont.)

TABLE 11-5 (Cont.)

Instruction	*Mnemonic*	*Operands*	*Op-code*	*Format*	*Explanation*
Subtract with Carry Halfword	SCH	R1, A(X2)	47	RX	The 16-bit second operand contents and the borrow from the previous operation are subtracted from the R1 contents. The result replaces the contents of R1 and reflects in the condition code setting.
Subtract with Carry Halfword Register	SCHR	R1, R2	OF	RR	
Compare Halfword	CH	R1, A(X2)	49	RX	The halfword second operand is compared to R1 contents. The result is indicated by the condition code setting.
Compare Halfword Register	CHR	R1, R2	05	RR	
Compare Halfword Immedite	CHI	R1, B(X2)	C9	RI	
Multiply Half word	MH	R1, A(X2)	4C	RX	R1 + 1 contents are multiplied by the contents of the second operand. The 32-bit result replaces R1 and R1 + 1. Both numbers are signed. R1 must be an even-number register.
Multiply Halfword Register	MHR	R1, R2	OC	RR	
Divide Halfword	DH	R1, A(X2)	4D	RX	R1 and R1 + 1 contents are divided by 16-bit second operand. The 16-bit signed remainder is stored in R1 and the 16-bit quotient is stored in R1 + 1. Both numbers are signed. R1 must be an even-number register.
Divide Halfword Register	DHR	R1, R2	OD	RR	



Exponent (64 notation)	*True HD exponent*	*True decimal exponent*	*Multiple fraction exponent*
00	−40	−64	16^{-64}
3F	− 1	− 1	16^{-1}
40	0	0	1
41	+ 1	+ 1	16
7F	+40	+64	16^{+63}

Notes: 1. The fraction is normalized if F_1 not zero

2. Since the fraction is only six digits, roundoff occurs

Operands	*After Equ.*	*Guard digit*
42134CD12	42134CD12	O
+4135678BC	+4201356 7B	C
	42148238E	E
	1	
	42148238F	

3. Convert fractional decimal to HD

.4062 67F

	.4062
	×16
6	6.4992
	×16
7	7.9872
	×16
F	15.7952

4. Convert HD to fractional decimal

$$4096\overline{)4062.0000000} \quad .4062$$

.4062 .67F

FIGURE 11-6. Floating-point formats (single precision).

operations keep track of sign; the fraction and exponent are in HD. Where the exponent is HD 40 the exponent will be 0. Above 40 we have positive exponents (for example, 41 equals an exponent of 1). Below 40 we have negative exponents (for example, 38 means an exponent of −2). The maximum exponents in HD are 7F. The first bit is the sign bit (1 for −, 0 for +). HD 7F equals 63 in base 10. HD 00 equals −63 in base 10.

11-6.2 Exponent Overflow and Underflow

When the resulting exponent is greater than 7F (63 decimal) overflow occurs, and the fraction and exponent bits are all set to 1. The normalizing process (moving the fraction so that no zeros appear directly after the decimal point) may provide an underflow. This is because the exponent exceeds −7F (−63 decimal). If underflow occurs, the entire result is set to true zero.

11-6.3 Guard Digit and Rounding

A guard digit is recorded to the right of the least significant HD digit. This digit does not appear in the final result but is used in intermediate processing. It does round off by adding 1 to the LSD for the HD fraction.

11-6.4 Floating-Point Instructions

The floating-point instructions are listed in Table 11-6. Where it is necessary to load floating-point registers, this is done by first loading into the general registers, as illustrated by the sequence of Problem 11-9.13 of Sec. 11-9.

11-7 STATUS SWITCHING AND INTERRUPTS

11-7.1 General

In the STOP mode the normal execution of instructions is suspended. The prcessor can be controlled via the display panel and the operator may examine the contents of general registers, the FP registers, and the PSW. The operator can execute instructions singularly or put the processor in the RUN mode. In the RUN mode the current PSW controls the operation. It can put the processor in the wait state, enable or disable interrupts, switch modes, and vary normal sequential constructions.

Bits 0-15 of the PSW are normally 0. These are the status bits. Bits 12-15 are the condition code. If X appears 1 or 0 is acceptable in bits 12-15. Bits 12-15 have a different meaning depending on whether the PSW is a normal one or one generated by a malfunctioning interrupt. The differences in bit meanings in both cases are listed as follows:

Normal PSW		*Malfunction interrupt PSW*					
12	(C) Carry or borrow	12	(C),	(V),	(G),	(L)	0 Power restore

13	(V)	Overflow	13	(V) = 1	Parity error on data read
14	(G)	Greater than >	14	(G) = 1	Parity error on instruction read
15	(L)	Less than <	15	(L) = 1	Power fails

Bits 16–32 are the present instruction counter.

11-7.2 PSW Bit Functions

Bit No.
0. *Wait state (W).* If (1) processor halts.
1. *External interrupt (EI).* If (1) processor responds to, request on multiplexor bus.
2. *Machine malfunction (M).* If (1) power fails or memory, parity error.
3. *Fixed-point divide fault (DF).* If (1) a quotient overflow occurs in a divide instruction.
4. *Auto I/O and interrupt mask (A).* If (1) these functions, enabled.
5. *FP interrupt mask (FP).* If (1) FP overflow, underflow, or, divide by 0.
6. *Q service interrupt mask (Q).* If (1) a queue requires service.
7. *Protect mode (P).* If (1) privileged instructions are not executed.

8., 9., 10., 11. } *0 when no condition code is enabled.*

12. C, 13. L, 14. G, 15. L } *= condition code*

Bits 16-31 are the instruction counter, as already pointed out.

11-7.3 Processor Interrupts

Interrupts cause the present PSW to be replaced by a new PSW. This breaks the normal instruction flow. During interrupt the old PSW is saved but replaced temporarily by the new PSW. This allows the normal program to be restored after a subroutine is performed.

11-7.4 Reserved Memory Locations

Locations below 100 are usually reserved for such items as interrupt pointers, the PSW, and system constants. Do not attempt to run programs at locations below 100.

11-7.5 Status Switching and Interrupt Instructions

These instructions are employed to manipulate the PSW generally. There is one instruction of this group (see Table 11-7) that will simulate an interrupt from a device.

TABLE 11-6 Floating-point instructions

Instruction	*Mnemonic*	*Operand*	*Op code*	*Format*	*Explanation*
Load Floating Point	LE	R1, A(X2)	68	RX	The FP second operand is normalized, if necessary, and placed in R1.
Load Floating Point Register	LER	R1, R2	28	RR	
Load Floating Point Multiple	LME	R1, A(X2)	72	RX	Successive FP registers, starting with R1, are loaded from successive memory locations starting with the address of the second operand. The process stops when FP register 14 has been loaded.
Add Floating Point	AE	R1, A(X2)	6A	RX	The exponents of the two operands are compared and the fraction of the smaller is shifted right four places at a time (1 HD digit), with the exponent incremented by 1 for each shift until the two are equal. The fractions are then added algebraically, and the results placed in R1. If addition produces a carry, the exponent of the results is incremented by 1 and the results are shifted right 1 HD digit. For subtraction the same processes are followed except that the difference between the second operand and R1 is placed in R1
Add Floating Point Register	AER	R1, R2	2A	RR	
Subtract Floating Point	SE	R1, A(X2)	6B	RX	
Subtract Floating-point Reg.	SER	R1, R2	2B	RR	
Compare Floating Point	69	R1, A(X2)	69	RX	R1 is compared to the contents of the second operand. Comparison is algebraic, taking account of the sign. The result is indicated by the

Compare Floating-point Reg.	29	R1, R2	29	RR	condition code setting
Multiply FloatingPoint	ME	R1, A(X2)	6C	RX	The exponents of each operand as derived from excess 64 notation is used in FP and added to produce the resulting exponent. The fractions are multiplied and
Multiply Floating-point Reg.	MER	R1, R2	2C	RR	the results rounded. If the FP value is not zero, the results are normalized and placed in R1.
Divide Floating Point	DE	R1, A(X2)	6D	RX	The exponent of each operand is derived from excess 64 notation used in FP. For division the exponents are subtracted. The first operand
Divide Floating-point Reg.	DER	R1, R2	2D	RR	fraction is is divided by the second operand fraction. Division occurs until the quotient is normalized, adjusting the exponent as required. No remainder is kept. R1 stores the quotient. The sign is per algebra rules.
Float Register	FLR	R1, R2	2F	RR	R1 specifies one of the FP registers R2 specifies the even number of an even/odd pair of general registers. The 32 bits contained in R2, R2 + 1 is converted to a FP number and stored in FP register R1. A full range of fixed-point numbers may be converted to FP. The fixed-point number 7FFF FFFF converts to FP 487F FFFF (largest positive), and fixed-point 8000 0000 converts to FP C880 0000 (the most negative). R1 results are normalized.
Fix Register	FXR	R1, R2	2E	RR	R1 specifies an even number of an even/odd pair of general registers. The FP number contained in the FP register is in two's compliment notation, whose integer value is obtained by shifting and truncating. The 32-bit result is stored in R1 and R1 + 1.

TABLE 11-7 Status switching and interrupts

Instruction	*Mnemonic*	*Operand*	*Op code*	*Format*	*Explanation*
Load PSW	LPSW	R1, A(X2)	C2	RX	The 32-bit second operand becomes the current PSW.
Exchange Program Status Register	EPSR	R1, R2	95	RR	Bits 0-15 of the current PSW replace R1. R2 replaces bits 0-15 of current PSW.
Simulate Interrupt	SINT	R, B(X2)	E2	RI	The LSBs of the second operand are presented to the interrupt handler as a device number. This number is used to index the interrupt service pointer, simulating an interrupt request from an external device.
Supervisor Call	SVC	R1, A(X2)	E1	RX	The address of the second operand replaces the contents of memory locations x0094–X0095. The current PSW replaces memory contents X0096–X0099. The halfword quantity in memory location X009A–X009B becomes the new PSW. The R1 field is doubled and added to contents of X009C and becomes the resulting address of the new location counter.

TABLE 11-8 Input/output instructions

Instruction	*Mnemonic*	*Operand*	*Op code*	*Format*	*Explanation*
Acknowledge Interrupt	ACK	R1, A(X2)	DF	RX	The address of the interrupting device replaces R1 contents. The 8-bit status replaces the second operand. The condition code is set equal to the device status byte.
Acknowledge Interrupt Reg.	ACKR	R1, R2	9F	RR	
Sense Status	SS	R1, A(X2)	DD	RX	Bits 8-15 of R1 address the device. The device 8-bit status is placed in the second operand location
Sense Status Register	SSR	R1, R2	9D	RR	
Output Command	OC	R1, A(X2)	DE	RX	Bits 8-15 of R1 address the device. The device transmits a command code via the second operand to the processor.
Output Command Register	OCR	R1, R2	9E	RR	
Read Data	RD	R1, A(X2)	DB	RX	Bits 8-15 of R1 address the device. The device via the second operand transmits a data byte to the processor
Read Data Register	RDR	R1, R2	9B	RR	
Read Halfword	RH	R1, A(X2)	D9	RX	Bits 8-15 of R1 address the device. The processor transmits a halfword to the device from the second operand location.
Read Halfword Register	RHR	R1, R2	99	RR	

(Cont.)

TABLE 11-8 (Cont.)

Instruction	Mnemonic	Operands	Op-code	Format	Explanation
Read Block	RB	R1, A(X2)	D7	RX	Bits 8-15 of R1 address the device. Bits 0-15 of the halfword located at the second operand is the starting address of the buffer. The processor transmits 8-bit bytes to consecutive buffer locations. The end of the buffer is specified by either the address of the second operand + 2 (A(X2) + 2 or R2 + 1).
Read Block Register	RBR	R1, R2	97	RR	
Write Data	WD	R1, A(X2)	DA	RX	Bits 8-15 of R1 address the device. The processor transmits a data byte from the second operand location.
Write Data Register	WDR	R1, R2	9A	RR	
Write Halfword	WH	R1, A(X2)	D8	RX	Bits 8-15 of R1 address the device. The processor transmits 16 bits of data as a halfword or two bytes from the second operand location.
Write Halfword Register	WHR	R1, R2	98	RR	
Write Block	WB	R1, A(X2)	D6	RX	Bits 8-15 of R1 address the device. Bits 0-15 of the second operand location contain the starting address of the data buffer. Bits 0-15 of the halfword located at the second operand address + 2 for R2 + 1 contain the ending address of the buffer.
Write Block Register	WBR	R1, R2	96	RR	
Autoload	AL	R1, A(X2)	D5	RX	This command loads the memory with bytes starting at location X0080 successively. The last byte is loaded at the second operand location.

11-8 SAMPLE PROGRAMS

11-8.1 General

This section generally demonstrates how one can prove that an instruction performs in the proper way. Also included are a few simple sample programs, and a program for using the display panel as a peripheral (an input device). These examples should allow students to prepare programs in assembler language, load them into the computer from the display panel, and thus demonstrate to themselves the operation, use, and power of such a processor.

11-8.2 Program Examples

This section lists eleven programs. The left-hand column lists the setting of the program counter. For example, Problem 11-9.3 starts at address 200. The next column to the right lists either the op code, memory address or data, depending on the instruction type (RX, RI, RR, or SF). In all problems we end in a loop. This is so at the run completion. The *Wait* light will come on if the program runs to completion properly. In the brackets at the right of the second column is listed the assembler mnemonic. Usually numbers involved in selected registers are listed on the extreme right. The sample problems include illustration of short operational programs like Problems 11-9.3, 11-9.12, or 11-9.13 or programs to illustrate the execution of a particular instruction, as illustrated in all other programs listed.

11-8.3 Simple Add-Subtract, Multiply, Problem

200	C810	LHI	
202	400		
204	C820	LHI	
206	500		
208	0412	AHR	$R_1, R_2 \rightarrow R_1$
20A	C830	LHI	
20C	100		
20E	OB13	SHR	$R_3 - R_1 \rightarrow R_1$
210	0871	LHR	$R_1 \rightarrow R_1$ & R_7
212	C850		
214	A		
216	OC41	MHR	$R_1 \times R_4 \rightarrow R_4, R_5$
218	4300	LOOP	
21A	218		

DECIMAL CALC

400	R_1	
+500	R_2	
900	R_1	
−100	R_3	
800	R_1, R_7	
A	R_4	
8000	(R_4, R_5)	5000 HD(R_4, R_5)

11-8.4 Fixed-Point Add Test Program

```
500   C820   Load Reg. 2 with next
             number.
502   1000
504   241A   Load A in Reg. 1
506   4010   Store Reg. 1 in next memory loc.
508   2000
50A   4A20   Adds Reg. 2 and what's in loc. 2000
50C   2000   Reg. 2 and loc. 2000 contain 100A
50E   4300
510   50E    Program loops
```

11-8.5 Divide (DH) Test Program

```
IC    Instruction or Data

700   C810 }                                  R1-200
702    200 } LHI 200                          R4-100 REMAINDER
704   4010                                    R5-3
706   1000
708   C85
70A    700
710   4D50   (DH)    DIVIDE LHI-700 BY 200
712   1000           STORE RESULTS
714   4300 ]
716    714 ] LOOP
```

11-8.6 Branch on Index High Test Program

```
IC    Instruction or Data

100   241A   LIS    ENTER A IN REG 1
102   C820   LHI    LOAD REG 2
104   FFFF    -1
106   2430   LIS    INITIALIZE REG 3
108   2442   LIS    MEMORY INCR. VALUE IN REG 4
10A   2450   LIS    INITIALIZE REG 5
10C   0861   BXH
10E   4065   STH    STR IN REG 6 WHAT's IN REG 5 (INITIALLY 0)
110   1000   STR    IN1000
```

BXH LOOP

```
IC      Instruction or Data

112     0A54    AHR     ADD REG 4 to REG 5
114     C010    BXH     ADD-1 to REG A AND CHECK AGAINST 0
116     010C    BALR
118     4300    B ┐     BRANCH
11A      118      ┘     to 108
```

STORED IN MEMORY THEN:

```
1000    A
1002    9
1004    8
1006    7
1008    6
100A    5
100C    4
100E    3
1010    2
1012    1
1014    0
```

11-8.7 Branch on Positive (BP) Test Program

```
IC      Instruction or Data

400     241E    LIS     LOAD REG 1 WITH E
402     242F    LIS     LOAD REG 2 WITH F
404     0A12    AHR     ADD R1 and R2          OB12  SUB R1 from R2
406     4220    BP      COMMAND
408      350            A + X2, ADDRESS 2ND OPERAND
40A     243B ┐          NO BRANCH LOOP               WHICH IT DID
40C     4300 │                                       REG 3-A
40E      40C ┘
350     243A ┐          BRANCH LOOP
352     4300 │          (WHICH IT DID)
354      352 ┘          REG 3  A
```

11-8.8 Compare Logical Halfword Test Program

```
IC    Instruction or Data

                                                    RESULTS
350   C810 }
352   0500 } LHI                          R1   500  500  500
354   C820 }                              R2   500  450  550
356   0500 } LHI                          R6   0    2    9
358   4020 }                          COND CODE      G > C
35A   1500 } STR 1500
35C   4510   (CLH)
35E   9566   (EPSR)   STR PSW COND
360   4300 ]          CODE
362   362  ] LOOP
```

11-8.9 Branch-on-not-Equal Register (BNER) Test Program

```
IC    Instruction or Data                          Changing these data

400   C810 } LHI
402   0200 }                                       402   200
404   C820 } LHI
406   0200 }                                       406   300
408   C840 }
40A   0350 } LHI
40C   0512   COMPARE CONTENTS OF R1 AND R2
40E   0384   BNER
410   243B ] LIS  LOAD REG 3
412   4300 ] NO BRANCH  WHICH IT DID
414   412  ] REG 3  B
350   243A ]
352   4300 ] BRANCH                                WHICH IT DID
354   352  ]                                       REG 3-A
```

11-8.10 Subtraction Halfword Reg (SHR) Program

```
IC    Instruction or Data

100   241F   LIS    LD WITH F      R1    1
102   241E   LIS    LD WITH E      R3    1
104   OB12   SHR    R1—R2
106   0831   LHR    PUT REG 1 IN REG 3
108   4300 ]
10A   108  ] LOOP
```

11-8.11 Branch and Return from Subroutine Program

IC	*Instruction or Data*	
600	241E	LIS ENTER E IN REG 1
602	242F	LIS ENTER F IN REG 2
604	C880	LHI
606	700	HW LOADED
608	0A12	ADD R_1 AND R_2—STORE IN R_1
60A	0148	BALR (60C STORED IN REG 4)
60C	243B	LIS ENTER B IN REG 3
60E	4300	] LOOP IF RETURN FROM SUBROUTINE
610	60E	]
700	2461	LIS ENTER 1 in REG 6
702	2472	LIS ENTER 2 IN REG 6
704	0A67	ADD R_6 AND R_7—STORE IN R6
706	0304	JUMP TO REG 4 (WHERE 60C IS STORED)
708	243A	LIS ENTER A IN REG 3
70A	4300	] LOOP IF DO NOT RETURN FROM SUBROUTINE
70C	70A	]
	REG 1	E + F IO
	REG 6	1 + 2 3
	REG 3	B (SUBROUTINE RETURNED TO 60C)

11-8.12 Program to Load Display with Bytes Test Program

IC	*Instruction or Data*	
100	2411	LIS
102	C830	} LHI—FOR INCREMENTAL LOAD
104	40	}
106	C8A0	} LHI
108	DD00	}
10A	C8B0	} LHI
10C	CC00	}
10E	C8C0	} LHI
110	BB00	}
112	C8D0	} LHI
114	AA0D	}
116	9E13	OCR
118	9A1A	)
11A	9A1B	} WDR
11C	9A1C	}
11E	9A1D	)
120	4300	} LOOP
122	120	}

AA	BB	CC	DD

PANEL DISPLAY

11-8.13 Load and Add Floating-Point Numbers

```
IC    Instruction or Data

100   C8C0  }
102   4223  }  LHI
104   C8D0  }
106   6200  }  LHI
108   C8E0  }
10A   4222  }  LHI
10C   C8F0  }
10E   3102  }  LHI
110   DOCO  }
112   1000  }  STM
114   6800  }
116   1000  }  Ld. FP
118   6820  }
11A   1004  }  Ld. FP
11C   2840     Ld. FP Reg.
11E   2A02     Add FP Reg.
120   4300  }
122   120   }  LOOP
```

Initially: FP Reg. 2 42 223102
FP Reg. 4 42 236200
Finally: FP Reg. 0 42 459302

11-9 SUMMARY

This chapter, after a short explanation at the block diagram level, continues on to describe in detail the available instruction repertoire. Since our model of the INTERDATA 7/16 did not have the double-precision floating-point option, we did not explain these particular instructions. A few sample programs were included to demonstrate how one can prove a particular instruction function. These are included in

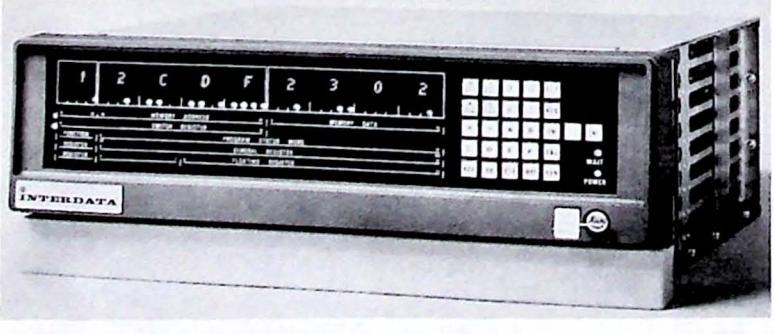


FIGURE 11-7. Interdata Model 7/16 Processor.

Sec. 11-9. Included here are a few figures supplied by INTERDATA Inc. These include Figs. 11-7, 11-8, and 11-9.

The carousel terminal is a 45-characters-per-second printer and comes in several optional widths, etc. A keyboard is included so that manual insertions can also be provided.

The disk in Fig. 11-9 occupies 7 in. of vertical space and records at 2200bpi.* The line printer employs dot matrix printing, as described in Chapter 6. The card reader reading speed depends on the stacker and is either 400 or 1000 CPM.†

*bpi = bits per inch.

†CPM = cards per minute.

11-10 PROBLEMS

1. Write a test program for fixed-point register multiply.
2. Write a program to load FP 040213A11 in R_1 and FP 0420012B1 in R_2. (Note that FP must be loaded from *general registers*.)
3. Subtract via a program R_2 and R_1. Write the answer that you would expect. (Use data in Problem 2 above.)
4. Write a test program for the BNER instruction.

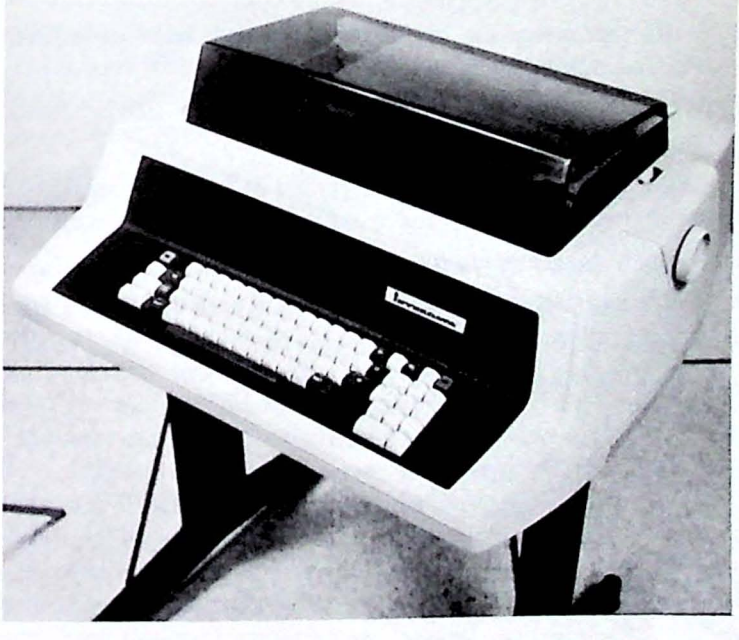


FIGURE 11-8. Perkin-Elmer Data Systems, Terminal Products Division, Carousel terminal.

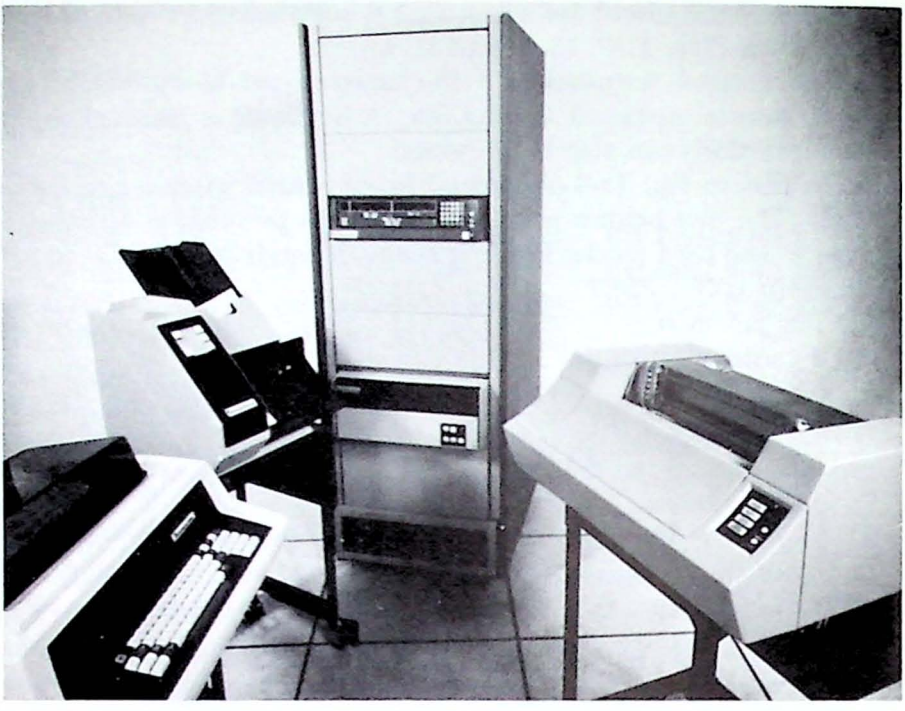


FIGURE 11-9. Interdata Model 7/32 Processor with 16 Megabit IOMB disc, carousel, card reader, and 60-200 LPM printer.

5. Write a program to load a printer with a byte of information AB. (Assume that the device address is 6.)
6. Write a program to load a halfword into the display (address= 1).
7. Write a program to branch forward in instruction BES. Show how to test it for both the *jump* and *none jump* conditions.
8. Write a test program for the BM instruction.
9. Write a test program to exclusive-OR two halfword registers.
10. If the halfwords for problem 9 are F3A2 and E1C8, what is the resulting output binary code expected?

12

Microprocessors and Microcomputers

12-1 INTRODUCTION

A microprocessor is an arithmetic logic unit (ALU) having the necessary adder, accumulator, program counter, stack registers, pointer control logic, control logic, and buses. Generally to make a microprocessor (μ-processor) into a microcomputer one must add ROM and RAM memory, interface logic elements, and a clock or clocks and power supplies. The stack registers are very much like those of the INTERDATA 7/16, having a stack pointer provided to address words in the stack. Since the field started in 1971 it has literally mushroomed. Four-bit processors started the game. These used BCD codes and originally appeared in hand calculators. Although people in he control field originally adopted these four-bit designs, the pressure was there for eight-bit and 16-bit devices. This was because most control applications use ASC11 codes and find it inconvenient and time consuming to employ up to three four-bit codes in sequence or more to move *op codes* and *addresses* about the ALU. Intel Inc. quickly developed their eight-bit model 8008 from their 4004, a four-bit μ-processor. Later the 8008 was upgraded in speed by a factor of 10 to 1 by going from P-MOS to N-MOS. This is the Intel model 8080, or now the 8080A. Motorola looked at the problem and developed its famous MC6800. Sixteen-bit designs, providing standard minicomputers, started as four four-bit or two eight-bit processors in parallel, but now are all on a single 16-bit chip. New technology is upgrading the speed of these μ-processors to bipolar speeds and I^2L devices will eventually exceed present bipolar speeds.

The original four-bit designs were designed for fixed programs. (An example would be a μ-processor used in a hand calculator.) But eight-bit and 16-bit designs now have abandoned this approach, and this is why many μ-computers still employ ROMs and RAMs as options to be added as needed. In early applications the μ-processors were absolutely limited to fixed applications because of mainly a ROM memory and a small scratchpad memory. Nowadays they normally work best in

applications where use of a minicomputer would be an overkill. However, even this reasoning is changing, since many of the modern minicomputers are built from μ-processors and supporting logic elements. For example, an overkill for a minicomputer could be, as an example, a traffic controller at an individual traffic light. One might very well use a minicomputer to monitor numerous individual light μ-processors. Thus, one can point to the merging of the minicomputer and microprocessor fields.

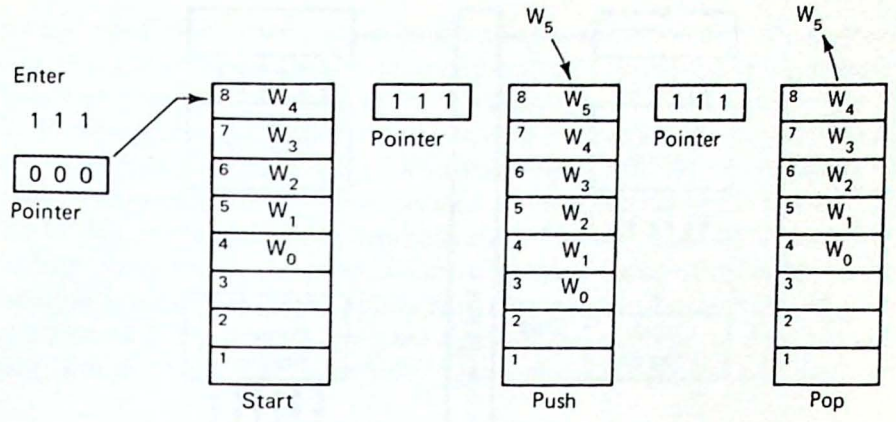
12-2 MICROPROCESSOR STACKS

Typical μ-processors employ stack registers. These have been only recent additions to the minicomputer field, such as the third generation designs. A stack is simply a sequential ordering of registers. Words may be entered at the top register and all other words below in the stack pushed down in sequence. Loading the stack is called the *push* operation. To remove a word a *pop* operation is performed. Here the word at the pointer is pulled out and all words below are pushed upward one position at a time. This form of stack has been named a last-in-first-out (LIFO) stack because of the push-pop operation.

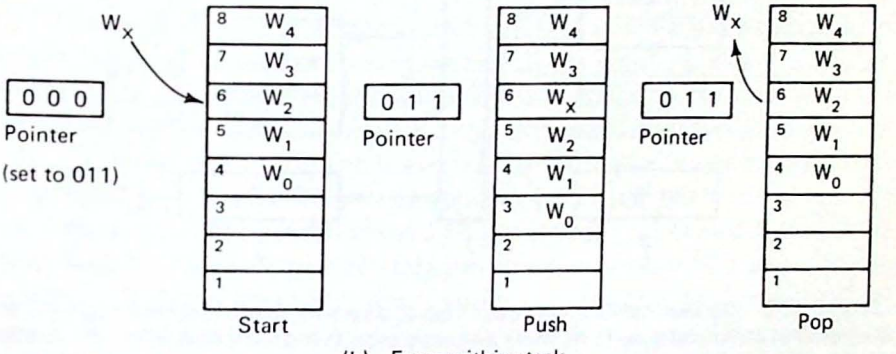
One other point concerns register addressing. For example, if a register consists of eight units, a three-bit stack address counter would be required. In most early μ-processors the stack held a list of sequential instructions. By changing the pointer address words are moved upward or downward in the stack. Normally a stack is charged with a great many words of data or instructions. When considering instructions only the programs are executed by popping up sequentially the stack stored instructions and transferring these sequentially to the control decoding matrix and the MAR.

Stack architecture eases the problems of *skips* and *jumps*. If, for example, an interrupt is received, a starting address for interrupt servicing is pushed into the stack and the P-C interprets the new instruction. [The program counter (P-C) has been called the instruction register (I-R) before.] The program address before the interrupt is pushed downward into the stack. In addition, the instructions for the interrupt servicing are pushed onto the stack. The interrupt program then proceeds by popping the stack. Once the interupt is serviced, the original P-C is at the top of the stack. This is incremented by 1 and the original program execution proceeds. Depending on the design of the individual μ-processor, stacks can be in hardware or memory. Hardware stacks are limited in length but are accessed rapidly. Memory stacks are longer, but access time is greater. The push-pull operation is illustrated by Fig. 12-1.

One other factor in LIFO action should be examined. Suppose that the pointer points to the fourth word down the stack, as in Fig. 12-1(b).



(a) From top of stack



(b) From within stack

FIGURE 12-1. Push and pop action for stacks.

The word pushed onto the stack will occupy the indicated position. The words above in the stack will stay in place, but the words below will be pushed down each one position. The reverse is true for the pop operation, and the word at the position pointer will be ejected from the stack. Above the position there is no movement, but below the ejected word each stack word climbs one position at a time. In this manner complete subroutines can be pushed onto the stack and executed in sequence, with a return to the original program after the subroutine completes its operation and is popped off the stack.

12-3 A TYPICAL MICROCOMPUTER

Figure 12-2 blocks out a typical μ-computer. The μ-processor itself provides: (1) machine control function, (2) program counter (formally IC), (3) the instruction register, (4) the address stack, (5) the arithmetic

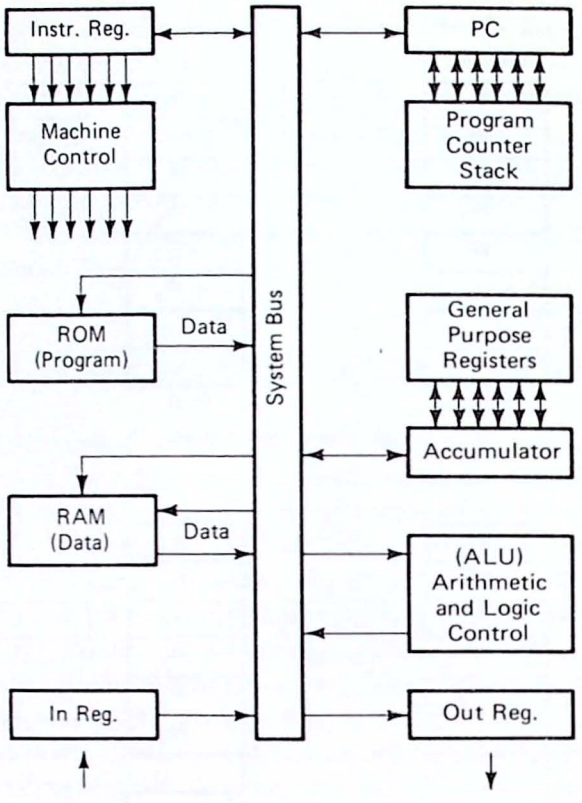


FIGURE 12-2. Generalized microprocessors block diagrams. Basic machine cycle-PC addresses next command code to ROM. ROM sends code to instruction register (IR). Control decodes IR and directs instruction execution. If memory reference instruction, RAM will be addressed. RAM will send or receive data from accumulator. If so instructed, ALU will alter data. These six steps generally are the sequence for a typical multiprocessor.

unit (ALU), and (6) the accumulator. Some processors have separate external buses for data movement between ROM, RAM, and I/O controllers and an address bus, as well as control lines (see Fig. 12-2). With a single system, bus data transfers, address transfers, ect. must be multiplexed on the single bus. This prevents simultaneous transfer of commands, addresses, and data.

12-4 MICROPROCESSOR FEATURES

Microprocessors are distinguished by their use of LSI technology. This approach has now been extended to supporting chips such as ROM, and RAM memory and peripheral interfacing chips. Some of the more common attributes are: *instruction sets* comparable with minicomputers; both *direct* and *indirect* addressing modes as well as *indexing* may

be provided. Generally stack addressing is included. *Software* is generally now about equivalent to minicomputers. *Interrupt capability* is important and generally extensive for the newer machines. *Execution time* for MOS designs is generally longer than for minicomputers, but with the use of silicon on sapphire and I^2L technologies this disadvantage will entirely evaporate. *Interfacing* should be compatible with bipolar TTL levels. Now special MSIs are provided. *Memory*, both ROM and RAM variety, is not usually part of the ALU chip. Quite often *timing* clocks, buffering, and power supplies must be supplied externally. In some cases small RAMs and ROMs are a part of the μ-processor chip, but usually, like the 8080A and M 6800, these memories are external chips.

12-5 INTEL 8080A

One of the most popular μ-processors since 1975 has been the Intel 8080 (now 8080A) as shown diagramatically in Fig. 12-3. The ALU handles eight-bit quantities via its adder and accumulator. Its stack registers are composed of three 16-bit registers. Because the ALU accumulator is only eight bits wide, the three general 16-bit registers are treated as six 8-bit registers. The processor generally operates in eight-bit bytes. All this data-flow is over bidirectional eight-bit paths. Data and instructions enter the ALU on an eight-bit data bus where instructions are sent to the *instruction register*, and there decoded for control. The processor controls data flow along internal bus paths. I/O data flow is via the accumulator and I/O buffer. Two 16-bit counting registers are on the chip (1) for program location and (2) as a push-pop stack counter. The P-C sends out a 16-bit address on the address bus to address memory modules and fetch the process instructions. The P-C is incremented by 1 each time it is used. The 8080 has 78 instructions at a recent count.

The processor organization and its functions are outlined below.

a. Register array and address logic. A Static RAM is provided to form either three 16-bit registers or six eight-bit registers. This permits an unlimited facility for subroutine nesting. Also provided is a 16-bit program counter (P-C), which essentially contains the current memory address, and a stack pointer, which addresses the next stack location in memory. Finally register pairs B-C, D-E, H-L, and W-Z are furnished. B-C, D-E, and H-L are three 16-bit registers (or six eight-bit registers) employed to hold data or addresses, etc. These registers are program addressable. Temporary register pairs W-Z are used for internal instructions and are not addressable. Sixteen-bit addresses from any of these registers (except W-Z can be incremented and put in the address latch to address memory.

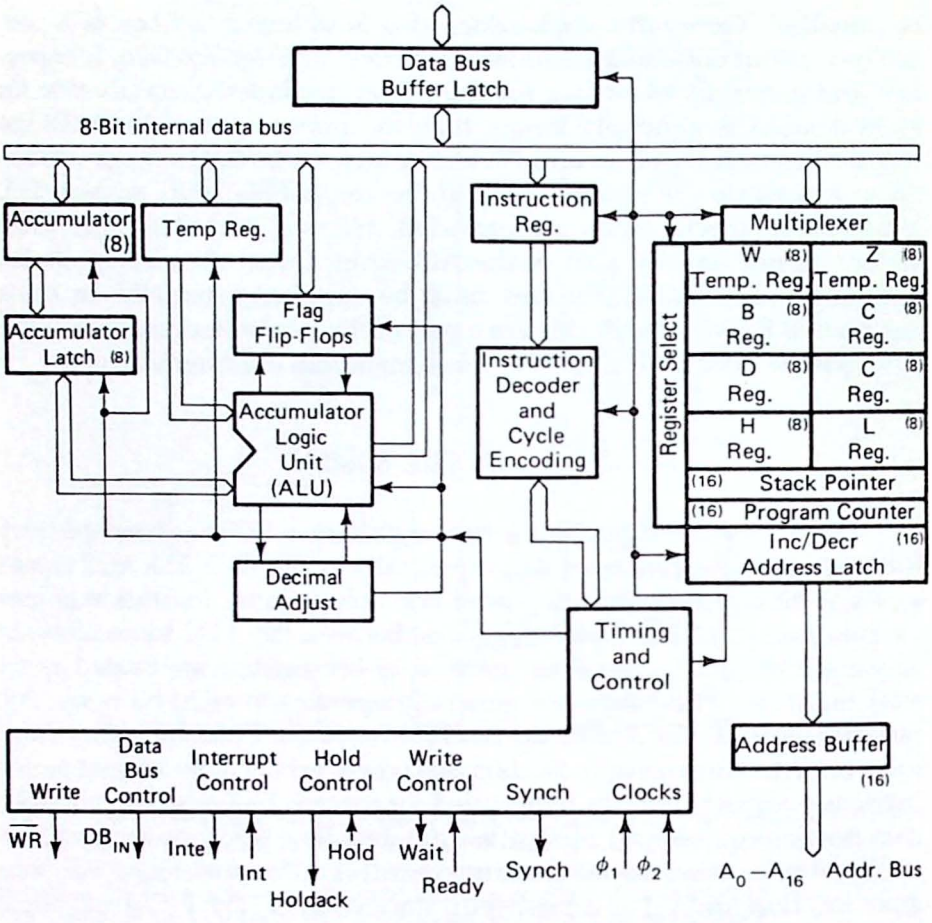


FIGURE 12-3. Intel 8080μ-processor block diagram.

2. Arithmetic logic unit (ALU). The ALU performs arithmetic, logical, and rotate functions. The components functioning within the ALU include an:

(a) eight-bit accumulator

(b) eight-bit temporary accumulator

(c) eight-bit temporary register, and

(d) a five-bit sign register

Decimal arithmetic is performed directly via a DAA instruction using auxiliary carries.

3. Instruction register (I-R) and control section. The I-R and the *timing and control* provides instruction op-code decoding. Inputs are from the Ø1 and Ø2 clocks, the *ready, interrupt, reset,* and *hold* control signal, as well as the INTE, NLDA, DB, SYNC, WR, and WAIT strobes.

4. Data bus buffer. The *data bus* is bidirectional and is buffered from the internal processor logic by the *data bus buffer*. This approach allows many devices and memory elements to interface with the processor without loading down its internal logic.

The instruction cycle performs in sequence the following steps:

(1) fetch,
(2) read memory,
(3) write in memory,
(4) read stack,
(5) write stack,
(6) input,
(7) output,
(8) interrupt, and
(9) halt.

No instruction will contain more than five of these steps.

12-6 MOTOROLA M6800

This is the microprocessor unit (MPU) that is in control of the entire microcomputer system. Its block diagram is given in Fig. 12-4 with the individual blocks performing the following functions. The output buffer addresses the RAM and the input/out devices via PLAs or other interfacing chips. The buffer is comprised of two eight-bit sets or a total of 16-bits addressing capability. On this buffer the processor impresses the instruction register, the program counter (H and L), the stack pointer (H and L), or the index register, if called for during the instruction execution; and added to the program counter or instruction register via the ALU. Thus, this summation may be used to address memory during an indexing operation. The bidirectional data interface is furnished by the data buffer. The accumulators are included as two eight-bit units—accumulator A and accumulator B. The condition code register generally handles carry-out sums during addition or borrows incurred during subtractions. An interrupt mask associated with this register controls the recognition of interrupt signals. The interrupt and restart logic, the timing clocks 01 and 02, the bus control and halt logic signals, the instruction decoder and other control signals are all interfaced with the instruction decode and control logic block.

The M6800 μ-processor ALU was the first one to have associated with it a complete family of related building blocks including memory and interface controllers. This approach was quickly followed by Intel and all other manufacturers. The distinguishing feature of this family in-

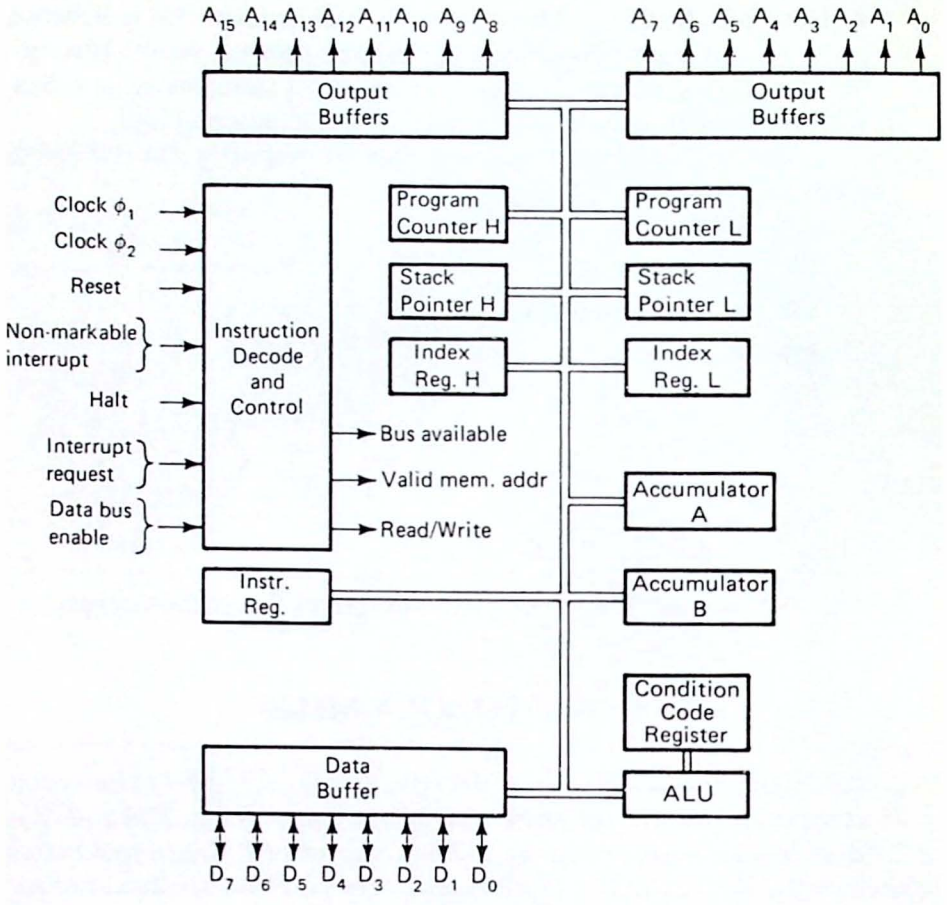


FIGURE 12-4. Motorola M6800 μ-processor block diagram.

cluded complete TTL compatibility, three buses that link all building blocks: (1) An eight-bit bidirectional data bus, (2) a 16-bit address bus, and (3) the 11-bit supervisory and control bus. An important concept typified by the M6800 architecture is the tying of the RAM and ROM memories and device controllers by these buses. Figure 12-4 summarizes in a block diagram form the μ-processor ALU architecture, while Fig. 12-5 blocks out the complete μ-computer system. Besides ROM and RAM memory the block functions are as follows:

PIA (parallel interface adopter). This is a dual I/O bus parallel interface device and permits a bidirectional data flow to the addressed device.

ACIA (asynchronous communication interface adopter). This block furnishes a serial interface to such devices as TTYs or modems.

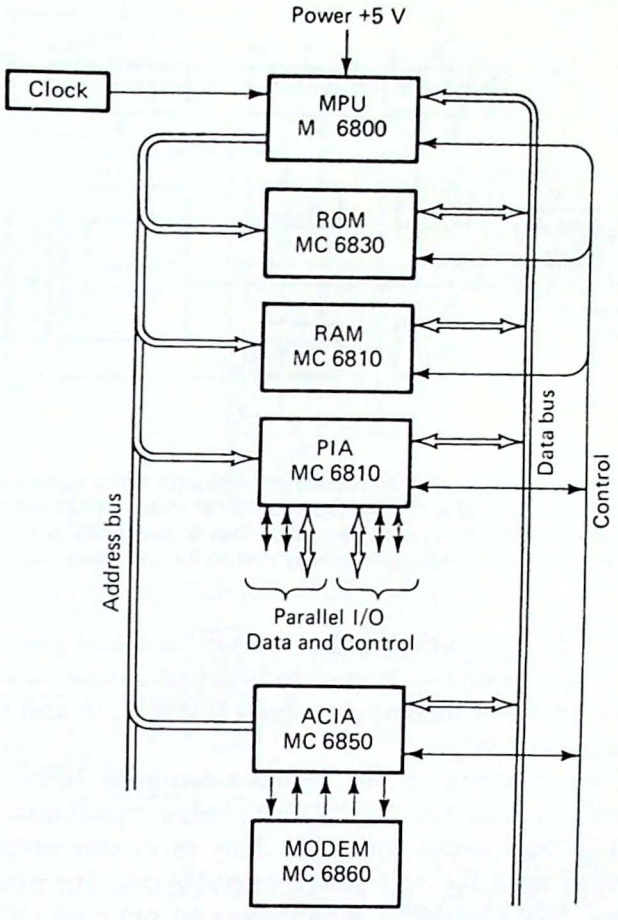


FIGURE 12-5. Motorola M 6800 μ-processor and supporting LSIs.

12-7 SIXTEEN-BIT MICROPROCESSORS

12-7.1 General

With bipolar (I^2L) logic μ-processor-based minicomputers are being constructed. Although Fig. 12-5 shows the original concept of paralleling up two eight-bit systems, there are now 16-bit μ-processors all intact being produced. Some examples are the MININOVA and National Semiconductor SC/M kit and the Texas Instruments 990 series.

The P-C counter provides a 16-bit address to memory. In the concept of Fig. 12-6, 16-bit words are interfaced with memory via the two eight-bit buffers. Control is via the I-R, μROM and array matrix. The pointer selects one of several words from the address stack. The instruction register supplies op codes to the array matrix, which selects microcommands from pROM, providing overall control to the entire sys-

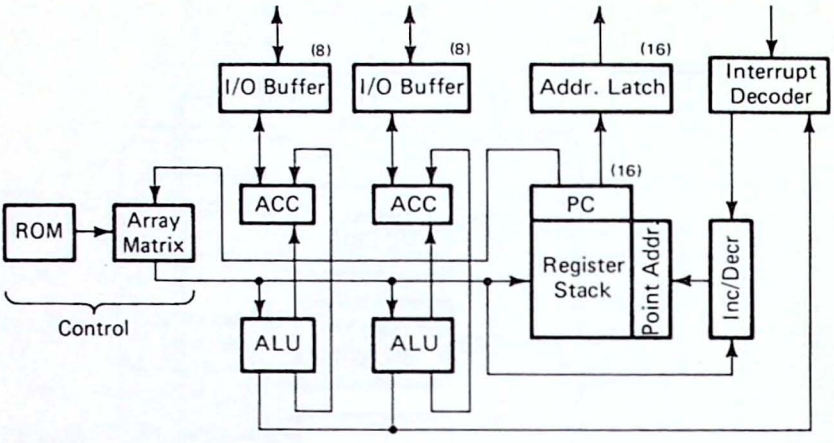


FIGURE 12-6. 16-bit μ-processor block diagram. Although this diagram shows effectively two 8-bit μ-processors in parallel, it is the practice now to make 1-chip μ-processors like the MicroNOVA and the Texas Instrument TM 9900. This is effectively a minicomputer when RAM and I/O interfaces are added, as they may well be for μ-processors.

tem including the interrupt decoder. All MSI and LSI packs probably fit on one or a few cards—a tremendous weight, size, and power improvement over the mimicomputers such as the H316 and Interdata 7/16 discussed in earlier chapters.

The Texas Instruments SBP 9900 is a complete 16-bit microprocessor with a 16-bit address bus and a 16-bit bidirectional data bus. It is I^2L, a bipolar logic compatible with TTL. This processor employs a single supply voltage, probably +5V, and is operable over the military temperature range (−55°Cto +125°C). It employs a 64-pin package. The internal logic of the chip includes the usual ALU, buffers, and a small ROM and RAM, with the majority of this type of logic on external chips not in the MPU.

12-7.2 A Sixteen-Bit Minicomputer of Microprocessor Components

Several manufacturers now provide 16-bit microprocessors that are the basic element of a microcomputer or a minicomputer. These are built in P-MOS, N-MOS, or I^2L logic chips. A typical organization (architecture) is illustrated by Fig. 12-7. The program counter (PC), and word pointer (WP) are still in the microprocessor element as are the ALU and status register, but missing are the word stack as internal registers. Instead, a portion of memory is designated as a work space, and 16-words in succession are designated by the WP as if they were internal hardware registers. Thus, 16-memory words become the internal registers as you

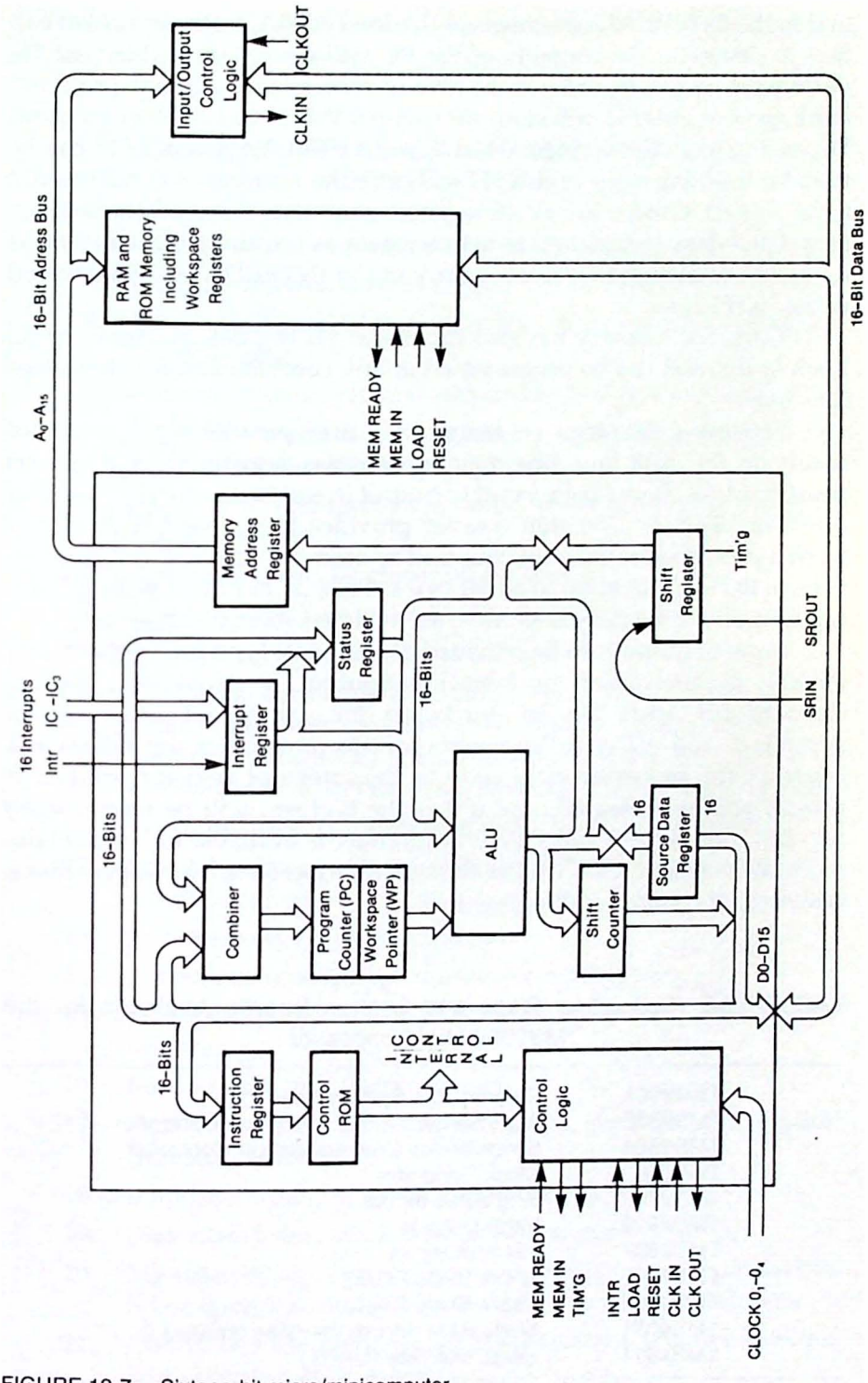


FIGURE 12-7. Sixteen-bit micro/minicomputer.

find in the INTERDATA machines or the Intel 8080A, both discussed earlier. A change to the contents of the PC will designate a different 16-word work space. In order to be able to restore the original program, work space register 13 will store the original WP, register 14 the original PC, and register 15 the original status word (SW). Registers 1–15 can be used for indexing while register O will store the number of bits a word is to be shifted. Otherwise, all 16 registers can store store addresses, accumulated data results, and temporary data as required. This is similar to the use of the internal register stack in the INTERDATA 7/16 covered in the last chapter.

Thus, the memory has two functions: (1) regular memory, or (2) work space, and can be programmed in any combination for these two functions.

The source data registers temporarily store parallel input or output words on the data bus. The memory address register (MAR) has its usual function. Data can be input or output in series via the shift register (SRIN or SROUT). The shift counter provides the capability to shift a word a prescribed number of counts. The memory can be any combinations of ROM or RAM up to 65.5K bytes. This, as has been pointed out, can be used for memory or work space registers in any sequence.

Input or output can be provided of most any type for a great many devices. Parallel 16-bit or 8-bit input/output is provided. In fact, standard I/O cards can be purchased for just about any type of peripheral, and A/D and D/A conversion. In most cases the device will interrupt the processor via a code to the interrupt register, where its priority will be evaluated, and if it is the highest, will be immediately serviced, otherwise it waits its turn. Interface is available for parallel devices like magnetic CRMT tapes or serial interfaces such as floppy disks, cassettes, or communication terminals.

TABLE 12-2 Supporting Chips and System Boards Available for the TMS9900 Microprocessor

TSM9901	Programable System Interface
TMS9902	Asynchronous Communication Controller
TMS9903	Synchronous Communication Controller
TIM9904	Clock Generator
TMS9905	Single Bus Inputs
TMS9906	Latch Outputs
TMS9907	Interrupt Inputs
TMS4042	256x4 Static ROM
TMS4043	256x4 Static RAM
TMS4051	Static RAM refresh for cycle stealing
TMS9911	Serial Interface (UART)

These are just a representative sample. There are others, but most of these presently use standard logic elements.

12-8 EXAMPLES OF MICROCOMPUTER APPLICATIONS

1. Process control equipment for data gathering and reduction.
2. Intelligent traffic control for trains, autos, and assembly lines.
3. Medical equipment from radiation measurement to laboratory analysis gear.
4. Measurements systems—sensors to flow process monitoring.
5. Educational systems—for example, microfilm reading/recording control and the control of libraries.
6. Automatic auto ignition systems, pollution control, and diagnostic systems.
7. Multiprocessing engineering systems.
8. Industrial appliances and utility output systems.
9. Multiplexers, modems, and communications control.
10. Intelligent terminals including display unit records and remote equipment. Also, terminal add/ons to give data to/from standard peripherals such as CRTs and teletypes. One typical use would be to integrate keyboards on a reservation system.
11. Small business computers of many varities and types.
12. Special-purpose terminals that will read badges and other similar devices.
13. I/O channels for large computers.
14. Automatic typesetting controls for newspapers.
15. Low-cost navigation equipment and integrating other military systems.
16. Stand-alone desk-top computer.
17. Point of sale terminals in stores and supermarkets. Optical character readers.
18. Banking terminals, chash dispensers, etc.
19. Automated time clocks and payroll systems.
20. Manufacturing control systems such as automated production lines, chemical and gas controllers, testers, and analyzers.
21. Controllers for buildings, such as elevators, security systems, and environmental control.

12-9 SUMMARY

This chapter first of all outlines the general features of μ-processors. It points to the use of BCD, octal, or hexadecimal coding providing usefulness in arithmetic problem solving and control. It was pointed out tha the original four-bit μ-processors were originally designed for BCD arithmetic, while eight- and 16-bit processors were best for control operations. Section 12-1 outlines the basic operation, while Section 12-3 describes briefly μ-processor minicomputers. Section 12-5 and and 12-6 describe two of the most popular μ-processors in use today. Table 12-1 outlines several typical μ-processors. Section 12-7 points to the newer 16-bit μ-processors originated in 1976 and 1977.

TABLE 12-1 Microprocessor characteristics

Manufacturer	*INTEL*	*MOTOROLA*	*NATIONAL SEMICONDUCTOR*	*DATA GENERAL*	*TEXAS INSTR.*
Model	8080A	MC 6800	SC/M	microNOVA	SBP 9900
Material	N-MOS	N-MOS	N-MOS	N-Mos	I^2L
Data bits	8	8	8	16/32	16
Mem bus	16	16	16	16	16
Instr.	116	72	46	—	69
P'k'g pins	40	40	40	40	64
Interrupts	1-line	2-lines	—	16 priority	16
Add time	2 μ-sec	2 μsec	—	—	3·5 μsec
Mem, cycle	700 nsec	700 nsec	—	—	135 n sec

12-10 PROBLEMS

1. In four-bit μ-processors the four-bit BCD code was popular. For what reason?
2. Why are eight-bit μ-computers popular for control applications?
3. Describe push-pop operation for a stack.
4. Why are 16-bit addresses (16-bit address bus) used in μ-computers?
5. C-MOS material is giving away to N-MOS for what reason?
6. Since ALU four-bit chips exist in TTL form, would it seem reasonable to expect many 16-bit μ-computers to use these to simulate a minicomputer? Give your reasons.
7. Motorola's M6800 led the field in what respect?
8. What is unique about the miniNOVA and SBP9900 μ-processors?

13

Typical Applications of Minicomputers

13-1 INTRODUCTION

This is a short chapter intending to show the wide application possibilities of minicomputers generally, although the Wall Street application is not strictly a total minicomputer system. This chapter is by no means intended to more than scratch the surface of possibilities, since the minicomputer users and uses continue to explode at a rate that defies definition, and is really beyond comprehension.

13-2 AUTOMATED TESTING

In the auto industry minicomputers at a variety of individual test stations (satellite) perform testing automatically on various components, processes, and assemblies. These satellites usuall report directly via wired lines to a larger computer for further data processing, or to receive control programs from stored data banks. The industry has not so far considered microprocessors, but these too will probably come. This general type of system is not unique.

Honeywell Inc., as well as many other manufacturers, has developed a complete line of computer-controlled test equipment, often throughout separate company divisions. Generally this equipment is referred to as *automated test equipment* (ATE).

Because the computer involved translates complex functions into relatively simple displays (panel lights, graphs, or CRT displays) and instructions, minimal skill levels are required to operate the test system. The operator merely connects the device to be tested, often via an adapter, to the test station and presses a BEGIN TEST button. This commands the test station computer to retrieve from its memory the specific test sequence required, and decision criteria for the system test involved. The sequence for a typical test follows:

- Check for proper interface connections to the tested item.

- Apply power to the tested item.
- Command stimuli application.
- Command measurement.
- Compute test data.
- Compare to decision limits.
- Announce or display decision.
- Log decision.

The testing sequence can evaluate the accuracy of the test station, and apply routines for isolating faults. If operator intervention is required or there are adjustments to be made, appropriate displays relate this information ot the operator. This method of testing is much faster than manual methods, if the volume of testing is sufficient to justify the cost of test equipment.

At Honeywell Aero-Florida several test stations were built for testing MM III component cards, memory, and the total MM III computer. These, and other specialized test equipment often included an H316 dedicated computer—that is, the H316 operated only that particular test station. Program loading was by high-speed paper tape reader, while test results were typed on a TTY. The eventual plan was to connect the individual test stations via a single wire bus to a central processor, much as the Minneapolis Aero ATE system, but because of funding limitations this plan was eventually abandoned.

The Minneapolis Areo ATE system connects up to 16 satellites (H316-controlled test stations) to a central DDP516. The total length of a single wire connection bus is 1000 ft. The CPU can thus talk to any satellite one at a time, but this originally was at a rate of 250K bits/sec. The satellites perform the required testing with control programs fed to them via the bus, and return reports over the same link, where the CPU can prepare recordings or printed records. A block diagram appears in Fig. 13-1.

13-3 THE DISNEY WORLD MONITOR

The Automated Monitor and Control System (AM & CS) of this famous attraction depends on a reasonably complex RCA-designed and -installed multiple minicomputer system. Six Data General Corporation Supernova computers (4K word memory machines) in turn communicate with 1300 sensors from six distributed control points. Important sensors, such as those for fire, are scanned separately by two Supernovas. If a sensor change is detected, this fact is forwarded to two re-

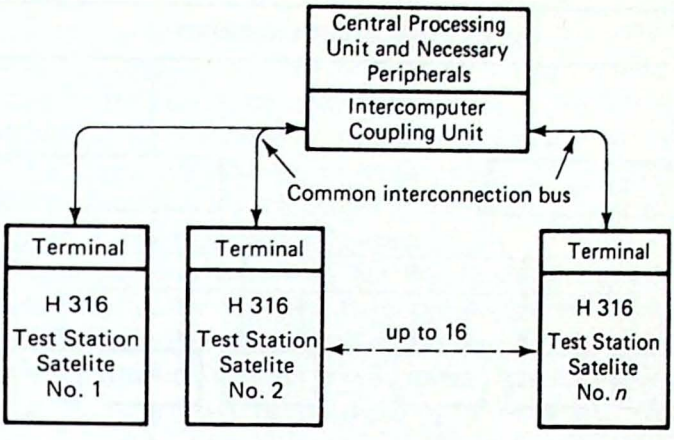


FIGURE 13-1. The Honeywell ATE system.

dundant 16K memory CPUs via duplex MODEMS and wire lines. These CPUs then operate appropriate displays, or transmit alarm information. Figure 13-2 block diagrams this system.

13-4 POWER SYSTEMS

13-4.1 General

Although computers are employed by electric utilities to scan loads on various lines and buses, and display and control various equipment condition, the primary use is analysis of power flow throughout the system. The effect of load, voltage, or equipment changes can be simulated or observed. The sudden loss of vital transformers, lines, generators, etc. can be observed as it affects transients at various points in the power network. The major effort here is to replace analog simulations with programmable digital programs.

13-4.2 Minicomputer Control of a Combined-Cycle Power Plant

A modern example of minicomputer control involves combined-cycle power plants, using gas turbines and steam technology, but employing gas converted from coal to fire the boilers. This system has many advantages over the conventional steam plants, including low capital investment cost per kilowatt of generation, low daily operating costs, and low atmospheric pollution. A standardized control system is avail-

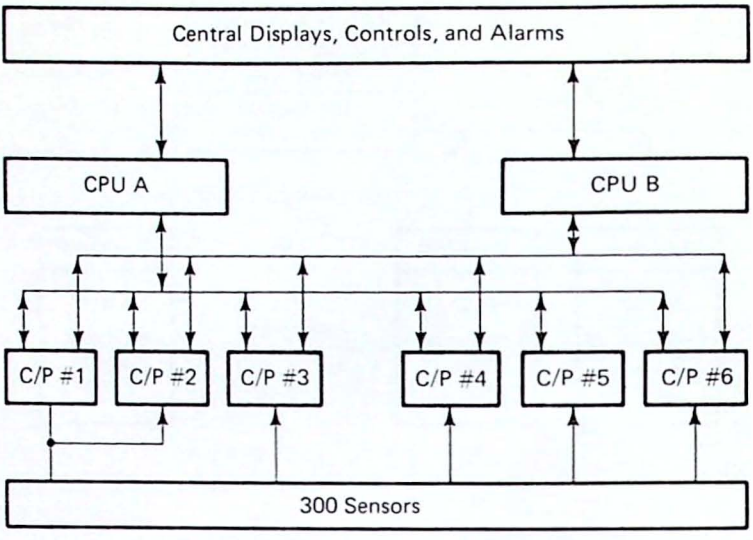


FIGURE 13-2. The Disney World AM & C system.

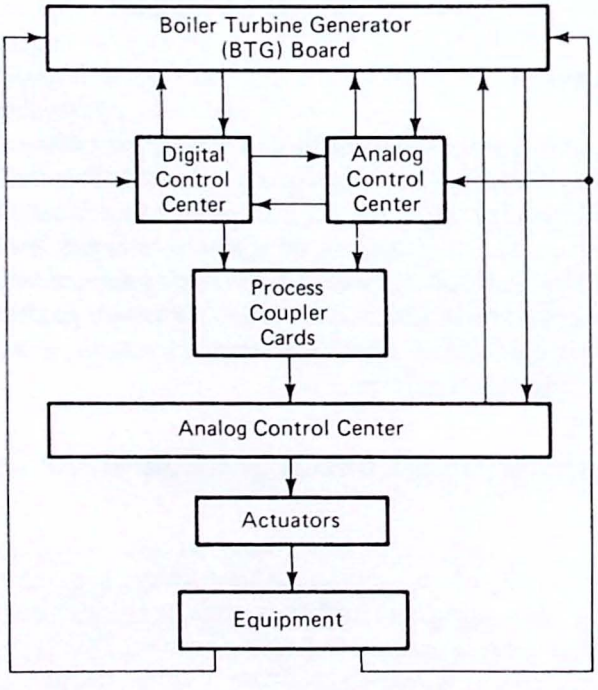


FIGURE 13-3. Combined-cycle power plant basic control system block diagram.

able for operating these plants, which starts all equipment, synchronizes the generators, controls the turbine blade positions, regulates the steam throttling pressure, loads the units automatically, monitors the line outputs, and provides for automatic shutdown when conditions call for. In October 1977 there were seven of these plants either on-line or in the planning stages in the United States and Mexico. Figure 13-3 shows a block diagram of the basic control system.

Analog control is necessary for the actuators controlling equipment. Digital control is supplied from the computers to interface with the analog circuits, operate the boiler-turbine generator board display, and accept digital-converted inputs from operational control and monitoring. There are two minicomputers in the system digital control center. One is assigned to all the control functions in the plant, whereas the second collects data, logs, monitors, and performs certain supervisory functions and computations. This second unit has a mass memory (disk) of 500,000 words for the storage of various files and program software. The two computers are connected via a data link, and control information is sent to the data computer once each minute for monitoring functions, logging and display purposes. The digital control center block diagram appears in Fig. 13-4 and functions generally as indicated above.

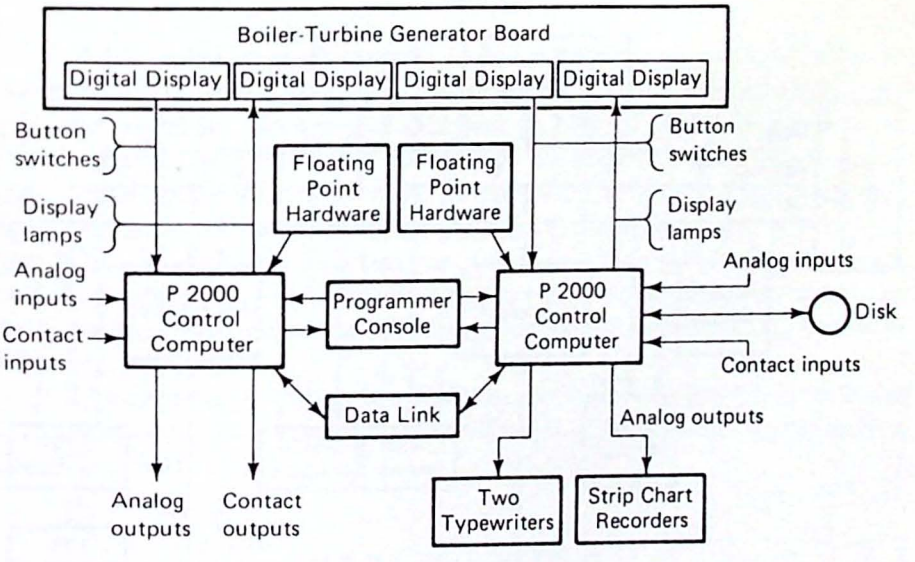


FIGURE 13-4. Digital control center block diagram. The explanation and these figures are from an excellent article appearing in the October 1977 *IEEE Spectrum*. For complete details the reader is referred to the article, "Computer Control of Combined-Cycle Power Plants," pages 36-41, by Robert Uram of the Westinghouse Electric Corporation.

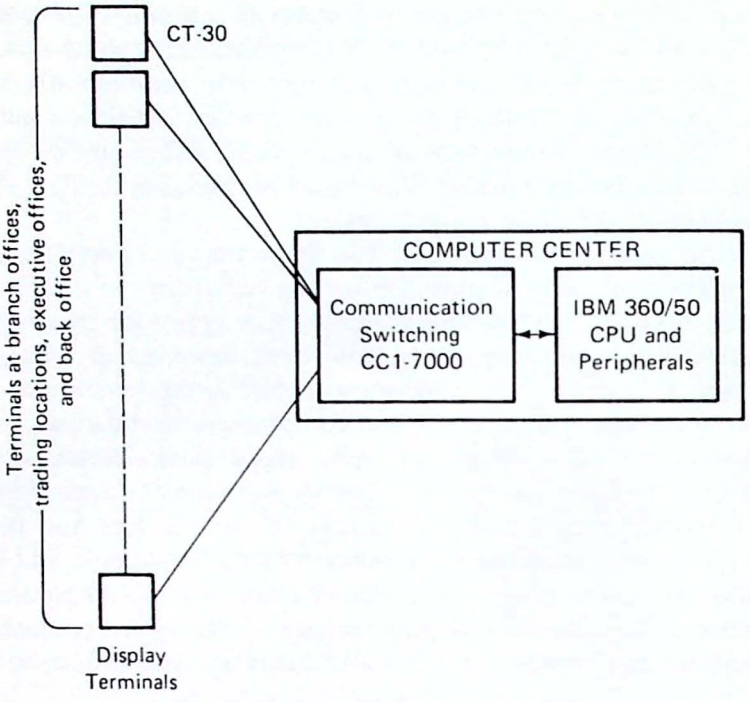

CT-30
Terminals at branch offices, trading locations, executive offices, and back office
COMPUTER CENTER
Communication Switching CC1-7000
IBM 360/50 CPU and Peripherals
Display Terminals

FIGURE 13-5. Signet 80 system.

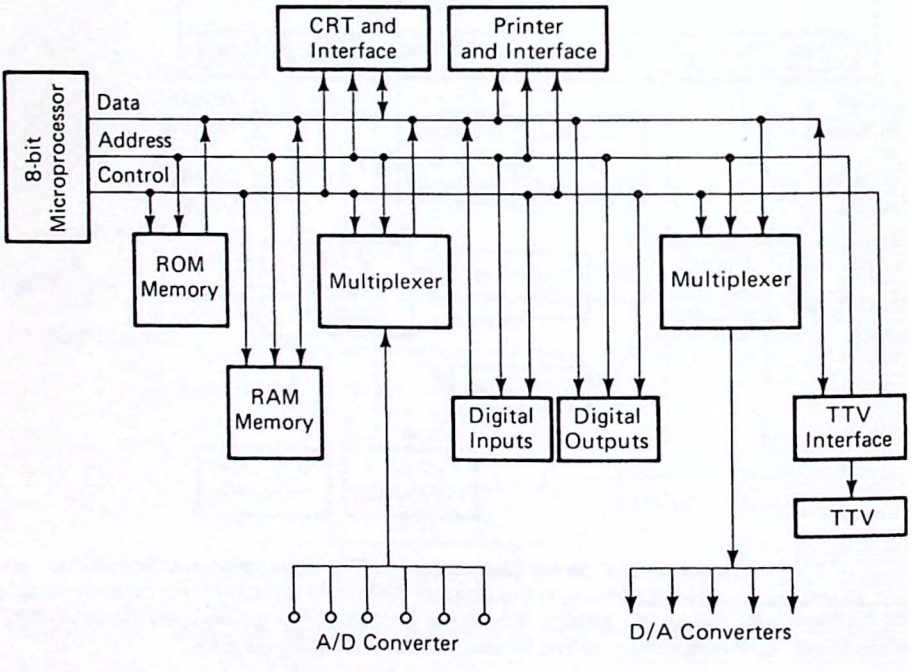


FIGURE 13-6. μ-computer control application.

13-5 WALL STREET AUTOMATION

A system (Signet 80) handles transactions involving stock trades for 30 or 40 brokerage firms at the local branch office, main office, and on the exchange floor. Two operators replace a chain of telephone operators, order clerks, messengers, etc. and thus reduce overhead considerably. The cost of the system is shared by all participants. Its cost savings is significant. Equipment includes:

1. CC-30 TV/Keyboard terminals—at branch offices and operational locations within the firms. TTYs enable permanent records to be kept.
2. CCI-7000 Communication processing systems provide message services between subscriber branch offices, trading locations, etc., employing telephone lines for up to 120 lines with up to 15 stations per line. The IBM 360/50 CPU is employed (hardly a minicomputer) for control, storage, analysis retrieval, and handling broker-oriented messages. The system employs Communication Inc. data transmission equipment. See Fig. 13-5.

13-6 MICROCOMPUTER/MINICOMPUTER APPLICATIONS

Because microcomputers and minicomputers are essentially equivalent devices, only a couple of applications will be outlined. For example, the eight-bit devices are excellent in control applications, because most control equipment and commands are eight-bit bytes. Such a typical control application is outlined in Fig. 13-6. Here D/A converters, A/D converters, as well as digital devices, are connected to the microcomputer. It has an eight-bit data bus, so byte commands are readily transmitted. Eight-bit A/D and D/A converters are connected via a multiplexor or by separate controllers. For recording purposes a printer, CRT, and teletype are included.

The second operation is a 16-bit microcomputer (effectively a 16-bit minicomputer). The example shown here is a data processing operation (see Fig. 13-7).

13-7 SUMMARY

Although applications of minicomputers could have referenced a number of other applications such as oil pipe-line control or hospital intensive care unit monitoring, only four examples (automated testing,

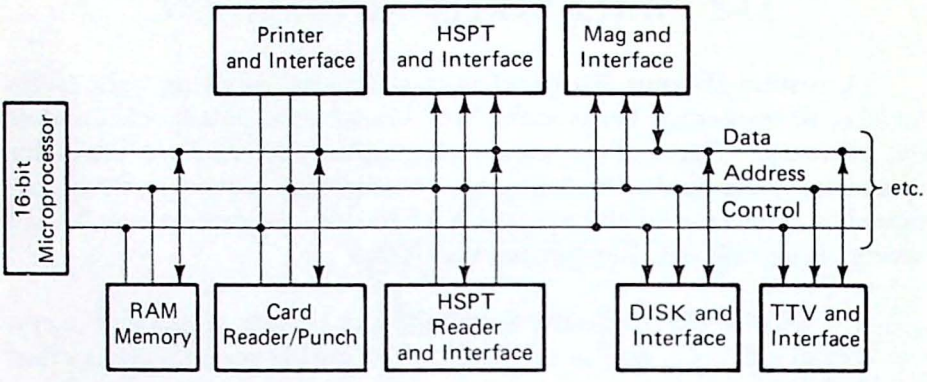


FIGURE 13-7. A μ-computer (minicomputer) data processing application. (In large applicatins buffers may be needed for I/O.)

Disney World monitoring, power system applications, and Wall Street automation) were chosen as representative uses. It should be evident that minicomputer usage is versatile and widespread, even exhibiting some complex system characteristics, although simple systems are likewise available.

13-8 PROBLEMS

1. What are the advantages in *automated testing*?
2. Describe the Honeywell ATE system.
3. What is the function of the Disney World monitor?
4. If the Signet 80 system were reconfigured, why do you think an INTERDATA 7/16 or similar computer could replace the IBM 360/50?
5. In what ways could a minicomputer be employed in a power station application?
6. What are some distinct advantages in minicomputers built from μ-processor hardware?

Appendix A

ANSWERS TO EVEN-NUMBERED PROBLEMS

Chapter 1

2. See Secs. 1-2.2, 1-2.3, 1-2.4, and 1-3.
4. Engineering or scientific problems involving differential equations.
6. *Batch processing* is covered in Sec. 1-5.1.
8. *Time-sharing* is covered in Sec. 1-5.3.
10. (a) A minicomputer is a small processor making use of MSIs and LSIs. They are parallel organized, making use of binary codes, usually octal or hexadecimal (HD), but *binary-coded decimal* (BCD) may be included and usually alphanumerics via ASC11 code. (SEE Chapter 2). (b) The invention of MSI and LSIs made minicomputers economically feasible.

Chapter 2

2. $.3030_5$
4. 101110
6. 1111000
8. 3875
10. C2E14

Chapter 3

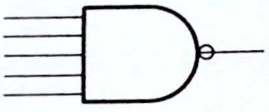
2. The op code, the variant bits, and the address (See Chapter 7 for memory reference instruction.)
4. *Memory.* To store temporary data and programs

Arithmetic unit. To perform computations
Control. To control the processor, memory, input/output, and peripherals via input/output
Input/output (I/O). To control and transfer data outside the computer.

6. (a) 24 μsec, (b) 1 μsec

Chapter 4

2. A NAND gate is a coincident gate, but its *true* output is low (0) rather than (1) for an AND gage.



4. An S-R flip-flop has a *set* and *reset* only. A J-K flip-flop has a *set* and *reset* also, but the inputs can be to J and K if the input is also clocked at the same time.
6. The changes should be: (1) eliminate the count 10 gating, (2) add one extra flip-flop stage.
8. This form of transmission is insensitive to external noise and changes in ground levels at either end of the line.
10. These are most probably used in I/O buffers.

Chapter 5

2. MBR. Transfer register between memory and CPU logic's buses.
MAR. Stores the address of the present memory word accessed.
Adder. Adds or subtracts numerical data or adds index register contents to the present MAR register contents for *indexing*.
Accumulator. Sums the adder inputs.
Instruction-counter (IC). Provides sequential address to the MAR.
Index register. Contents sums with present instruction address to change address before execution.
4. 6, .500000
6. 10, .000503, 13, .30120, adds to 13, .301703
8. (a) (1) IC addresses memory; (2) word is transferred to MAR

and *op-code decoder*, (3) test for *indexing* or *indirect addressing*. (b) *Adding index register* to present address called from memory.

10. a. Routes the address about the processor from the MAR, MBR, *Adder, I/O register*, etc.
    b. Provides data flow to or from peripherals.
    c. Interprets the *op-code* of the instruction.
    d. Determines if the peripheral is on or off and ready to transfer data.

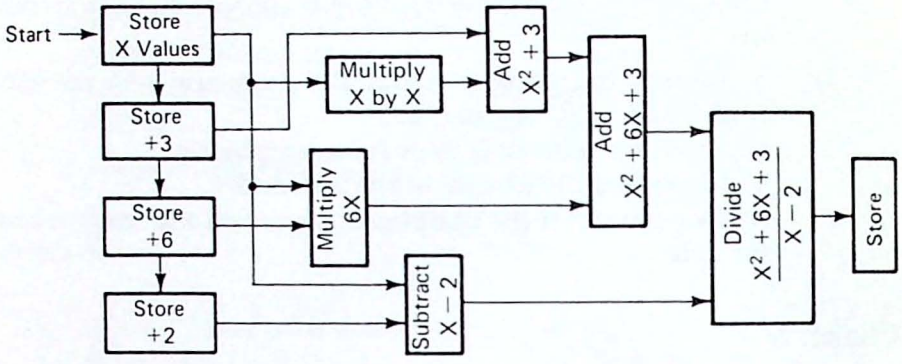
Chapt. 6

2. High-speed paper tape readers and punches. The tapes can be read on a CRMT unit.
4. CRMT units can store a great deal of data, but access to a particular message somewhere away from the present tape position takes a great deal of time to access. This is the most inexpensive form of mass storage on a per bit basis.
6. Primarily bulk storage is their task. Disks are read in parallel while finding the start of data on a tape requires searching for it serially on the tape. On a disk file data words are available after the short time required to move the heads a few inches. After that head movement to get to a word takes approximately 1/120th. of a second on the average.
8. To move two track requires approximately 20ms plus about 0.1ms on the average to move ½ way around the track. Ans. (20.01 ms.)
10. 50/5 = 10 μsecs.

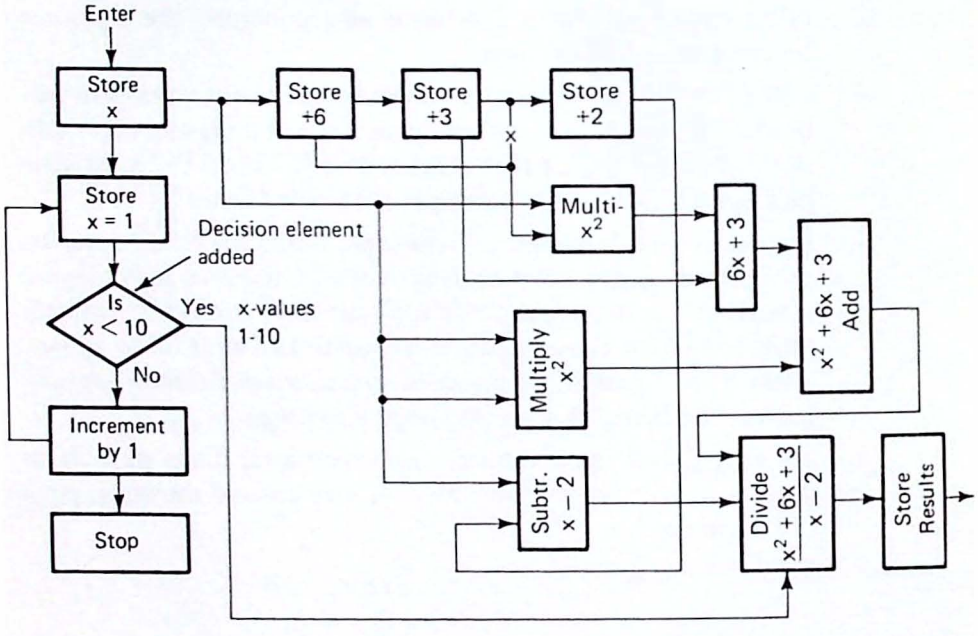
Chapt. 7

2. Shift, right, shift left, rotate left, rotate right. Bits 1-6 are op-code and bits 7-16 specify the amount of shift.
4. 10 bits for each address allowing 1024 words to be addressed

6.



8.



10. A subroutine is a fixed collection of instructions inserted in a program to do a specific task. *Linking statements* allow entering or leaving a subroutine and returning to the main program.

Chapter 8

2. operational program. One writen by the user to do a specific task.

executive program. One generally supplied by the manufacturer to take charge of other programs and allocate them in a specified priority sequence.

utility programs Those usually supplied by the manufacturer to do specific standard tasks, such as special arithmetic operations, or to operate peripherals or subroutines.

4. bootstrap load. Used to initially load the loading programs into the processor.

absolute. Loads starting at a specific address.

relocatable. Loads into memory without the programmer's selecting the starting address. This is done by the executive.

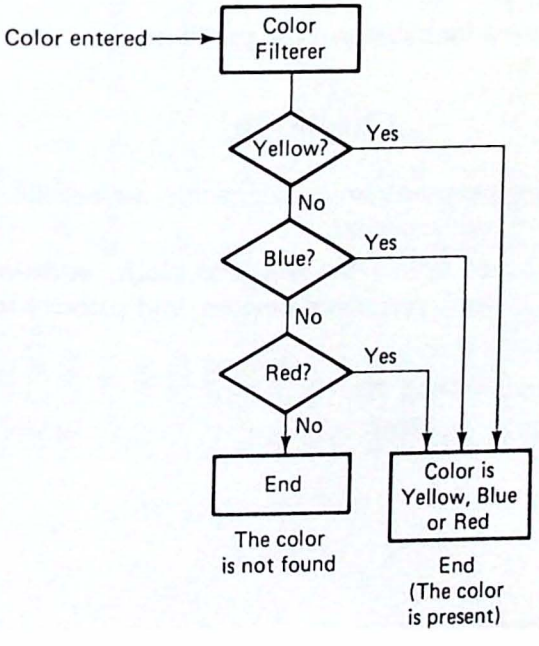
6. See Table 8-2.

8.

```
LOOPS 3 TIMES
    ┌──► 10 READ R
    │    20 DATA 3, 6, 12
    │    30 LET Y = 3.1416
    │    40 LET A = YR↑2
    └─── 50 GO TO 20
         60 PRINT A
         70 END

         To run program the operator types
         RUN on the keyboard
```

10. Is it red, blue, or yellow flow chart



Chapter 9

2. .267 − E4
4. n FORMAT (s_1, s_2 s_3, . . . , s_m) where n is the statement number and the s's are an ordered set of abreviations to describe datum field.
6. COMPUTE AREA OF RECTANGLE

```
1  READ (2,1) W, H
2  FORMAT (F4.1, F4.1)
3  ASSIGN 20 to W
4  ASSIGN 10 to H
5  GO TO 1
6  A = W × H
7  FORMAT (4.1, F4.1, F5.1)
8  PRINT W, H, A
9  STOP
10 END
```

8. GO TO Statement 1 when $K = 1$, GO TO statement 10 when $k = 2$, GO TO statement 12 when $k = 3$, and GO TO statement 13 when $k = 4$.
10. Most used for business-type problems.

Chapter 10

2. Loading programs, *utility subroutines*, an *assembler*, an *executive program*, and a *compiler*.
4. High-speed arithmetic, real-time clock, additional memory, DMC control, peripheral devices, and priority interrupts.
6. See Table 11-1.
8. 1 101 111 010 001 011
10. 00101011, 01001101

2.

100 C810 102 4021	Ld. Rg. 1	11A C850 11C 4200	Ld. Rg. 5
104 C820 106 3A11	Ld. Rg. 2	11E C860 120 12A1	Ld. Rg. 6
108 C830 10A 4200	Ld. Rg. 3	122 4050 124 2000	Move Rg. 5 to mem.
10C C840 10E 12B1	Ld. Rg. 4	126 4060 128 2002	Move Rg. 6 to mem.
110 4010	Move Rg. 1 to mem.	12A 6820	Ld. FP Rg. 2
112 1000		12C 2A12	Add FP R. 1 and Rg. 2
114 4020	Move Rg. 2 to mem.	12E 4300 130 12E	Ans. 40,33EB11
116 1002			

118 6810 Ld. FP Reg. 1

4.

400 C840 402 900		900 C830 902 A	Loop	Program if jump executed
404 241E	Ld. Rg. 1	904 4300		
406 242E	Ld. Rg. 2	906 904		
408 0B12	subtr.			
40A 0314	BNMR			Program jumps, so $R_3 = A$
40C C830 40E B 410 4300	program if next instruction executed			If E and F interchanged $R_3 = B$

6.
```
412 410 ┐  Loop
200 2411┘  (LIS)
202 C830   to put in
           normal mode

204 80

206 9E13 (OCR)
208 D810   (WH)
```

```
20A  400   addr.
400  4321  # loaded

402  4300
404  402   loop
```

8.
```
400  241F  Ld. Rg. 1
402  242E  Ld. Rg. 2
404  0B12  (subtr)
406  4210  BM instr.
```

```
900 243A ┐
902 4300 │
904 902  ┘
```

Program if jump Reg. 3 = A

If E and F are interchanged in Reg. 1 and Reg. 2.

```
408  900   2nd. addr.
40A  243B ┐
40C  4300 │  Program if next
40E  40C  ┘  instr. executed
```

Reg. 3 = B

10. 1110 1101 1001 0101

Chapter 12

2. Control applications often use ASC11 codes. Eight-bit transfers can be provided directly.
4. To get address capability to 64K bytes or words.
6. Yes; this is being done by companies such as Fairchild, etc. Also, 16-bit machines have been configured out of four 4-bit chips.
8. These have 16-bit I/O buses.

Chapter 13

2. Shown in Fig. 13-1 and described in Sec. 13-2.
4. The modern minicomputer can now handle processing of large machines. The INTERDATA 7/16 is for most purposes a 32-bit computer with lots of power and capability.
6. They are much smaller, and their price is bound to decrease drastically.

Appendix B

CHARACTERISTICS OF 8080A SYSTEM

1. *Instruction word length and cycle time*

(a)	8 bits	(op code)	1 cycle
(b)	16 bits	(op code 8-bit address)	2 cyles.
(c)	24 bits	(op code 16-bit address)	3 cycles

2. *Timing intervals and functions occurring*

(a)	T_1	Memory address or device number is placed on address bus.
(b)	T_2	The CUP samples *ready, hold* inputs, and *halt* instructions.
(c)	T_w	Optional period for CPU to enter *wait* state if *ready* is low, or if *halt* instruction is executed.
(d)	T_3	*Read* or *write* occurs.
(e)	T_4, T_5	Internal operations; may be skipped if instruction execution does not require.

3. *Strobe function actions*

(a)	MEM-R	Memory reads
(b)	MEM-W	memory writes
(c)	I/O-R	I/O devices receive data
(d)	I/O-W	I/O devices output data

4. *Sample supporting LSIs for 8080A system*

	LSI #	*Function*	*Provides*
(a)	8224	Clock generator and driver	Clock to CPU

(b)	8228	System controller, bus driver	Data and control drive
(c)	8702A	ROM	Read only memory
(d)	8222	RAM	Random address memory
(e)	8251	Communication interface	Serial adapter
(f)	8255	I/O peripheral interface	Parallel adapter
(g)	8214	Priority interrupt	Optional interrupt capability
(h)	8212	Address buffer	Optional

Appendix C

BIBLIOGRAPHY

Chapters 1, 2, and 3

Henzel, R. A., "Some Applications of Minicomputers (ARPA Network), *Computer (IEEE)*, (Sept.–Oct., 1971), pp. 19-21.

Eadie, D., *Introduction to the Basic Computer*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1968.

Chapter 4

Deem, Muchow, and Zeppa, *Digital Computer Circuits and Concepts*, 2nd ed. Reston, Va.: Reston Publishing Co. Inc., 1977.

Introduction to the Basic Computer (see above).

Chapter 5

Freedman, M. D., *Principles of Digital Computer Operation*. New York: John Wiley & Sons, 1972

H316 Input/Output Manual. Minneapolis, Minn.: Honeywell Inc., 1968

Chapter 6

Eadie, D., *Modern Data Processors and Systems*. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1972.

Manufacturer product bulletins.

Chapters 7 and 8

Crawford, F. R., *Introduction to Data Processing,* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1968.

Walnut, F. K., *Introduction to Computer Programming and Coding.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1968.

H316 Programmer's Reference Manual. Minneapolis, Minn.: Honeywell Inc., 1968.

DeRossi, C.J., *Learning BASIC Fast,* Reston, Va.: Reston Publishing Company Inc., 1974.

Chapter 9

Awad, E. M., *Automatic Data Processing, 2nd ed.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1970.

Anderson, D. M., *Computer Programming—FORTRAN IV.* New York: Appleton-Century-Crofts, 1966.
Chapter 10

Chapter 10

H316 Operator's Guide. Minneapolis, Minn.: Honeywell Inc., 1968.

H316 General Description, Minneapolis, Minn.: Honeywell Inc., 1968.

Chapter 11

16-bit Processor User's Manual. Oceanport, N.J.: Interdata Inc., 1976.

Model 7/16 Processor Product Bulletin. Oceanport, N.J.: Interdata Inc., 1973.

OEM Catalog. Tinton Falls, N.J.: Perkins-Elmer Data Systems, 1977

Chapter 12

Hilburn, J. and Julie, P., *Microcomputers/Microprocessors: Hardware, Software, and Applications.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1977.

Klingman, E. *Microprocessor System Design.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1977.

Soucek, P., *Microprocessors & Microcomputers.* New York: John Wiley & Sons, 1976.

Cushing, R., *EDN's Third Annual Microprocessor Directory.* EDN (Nov. 20, 1976), pp. 44–89.

Davis, S., "A Fresh View of Mini and Microcomputers," *Computer-Design, May 1974, pp. 67–79.*

Chapter 13

"Eight-minicomputer System Controls Disney World Complex," *Computer-Design,* April 1973, pp. 46–48.

Sass, F.N., "Wall Street Automation—Real-time Trade Processing," *Telecommunications,* June 1973, pp. 48–50.

Kilin, T. and Tullos, R. E., "Automatic Test Systems," *IEEE Spectrum,* Sept. 1974, pp. 44.

Tinney, W. F. and Enns, M. K., "Controlling and Optimizing Power Systems," *IEEE Spectrum,* June 1974, pp. 56–60.

Uram, R., "Computer Control of Combined Cycle Power Plant, *IEEE Spectrum,* October 1977, pp. 36–41.

Appendix D

8080A INSTRUCTION SUMMARY

Mnemonic	*Bytes*	*Cycles*		
STA	3	4	Store A contents in memory addressed by byte two and three of the instruction	
LDA	3	4	Load A with contents of memory addressed by byte two and three of the instruction	
ADD M	1	2	$A \leftarrow A + M$	ADD
ADC M	1	2	$A \leftarrow A + M + C$	ADD with carry
SUB M	1	2	$A \leftarrow A - M$	SUBTRACT
SBB M	1	2	$A \leftarrow A - M - B$	SUBTRACT with borrow
ANA M	1	2	$A \leftarrow A \wedge M$	Logical AND
XRA M	1	2	$A \leftarrow A \veebar M$	Exclusive OR
ORA M	1	2	$A \leftarrow A \vee M$	Logical OR
INR M	1	3	$M \leftarrow M + 1$	Registers H and L are incremented by 1
DCR M	1	3	$M \leftarrow M - 1$	Registers H and L are decremented by 1
ADI M	2	2	$A \leftarrow A + B_2$	ADD
ACI M	2	2	$A \leftarrow A + B_2 + C$	ADD with carry
SUI M	2	2	$A \leftarrow A - B_2$	SUBTRACT
SBI M	2	2	$A \leftarrow A - B_2 - B$	SUBTRACT with borrow
ANI M	2	2	$A \leftarrow A \wedge B_2$	Logical AND
XRI M	2	2	$A \leftarrow A \veebar B_2$	Exclusive OR
ORI M	2	2	$A \leftarrow A \vee V B_2$	Logical OR
CPI M	2	2	$A - B$	Compare
LXI B*	3	3	$C \leftarrow B_2$; $B \leftarrow B_3$	Load byte 2 of instruction into C and byte 3 instruction into B.

**Similar instructions LXI D and LXI H also exist.*

In addition the the immediate addressing instructions listed above, a similar set of instructions exists for reference to the registers. There are

17 of these instructions employing one byte and one cycle. In addition to the ones that operate with the registers in a manner similar to the *immediate addressing instructions*, included within this group there are: RAL, rotate register A left 1 bit; RAR, rotate register A right 1 bit. RLC and RRC are similar but differ slightly on how they handle the carries and over flows. CMA complements A, STC sets the carry flip-flop to 1, and CMC complements the carry flip-flop.

Move data instructions

Mnemonic	*Bytes*	*Cycles*	*Operation*	
MOV r_1, r_2	1	1	$r_2 \leftarrow r_2$	
MOV r_1, M	1	2	$r_1 \leftarrow$ M	
MOV M, r_1	1	2	M $\leftarrow r_1$	
MVI r	2	2	r $\leftarrow B_7$	Load instruction byte 2 into r.
MVI M	2	3	M $\leftarrow B_2$	Load byte 2 of instruction into memory addressed by H and L.

Jump instructions

JMP	*3*	3	PC $\leftarrow B_3, B_2$	Unconditional jump to memory addressed by byte 2 and 3
JC	3	3	If C = 1, PC $\leftarrow B_3, B_2$.	Otherwise PC $\leftarrow$ PC + 3.
JNC	3	3	If C = O, PC $\leftarrow B_3, B_2$.	Otherwise PC $\leftarrow$ PC + 3
JZ	3	3	If 0 = 1, PC $\leftarrow B_3, B_2$.	Otherwise PC $\leftarrow$ PC + 3
JNZ	3	3	If 0 = 0, PC $\leftarrow B_3, B_2$.	Otherwise PC $\leftarrow$ PC + 3
JP	3	3	If S = 0,PC $\leftarrow B_3, B_2$.	Otherwise PC $\leftarrow$ PC + 3
JM	3	3	If S = 1, PC $\leftarrow B_3, B_2$.	Otherwise PC $\leftarrow$ PC + 3.
JPE	3	3	If parity = 1, PC $\leftarrow B_3, B_2$.	Otherwise PC $\leftarrow$ PC + 3.
JPO	3	3	If parity = 0, PC $\leftarrow B_3, B_2$.	Otherwise PC $\leftarrow$ PC + 3.

Conditional subroutine call and return instructions

CALL	3	5	[SP − 1] [SP − 2] ← PC, SP = SP − 2 PC ← B_3, B_2. Transfer PC contents to push-down stack in M addressed by SP. SP is decremented by 2. Jump unconditionally to instruction located in M by byte 2 and byte 3 of instructions.
RET	1	3	PC ← SP, [SP + 1], SP = SP + 2 Return to the instruction in M location addressed by last values shifted onto stack addressed by SP. SP incremented by 2.

There are eight more call instructions of 3 bytes and 3/5 cycles and nine more return instructions of 1 byte and 1/3 cycles. (Refer to 8080A manual or text giving total instruction details.

Input/output instructions

IN	2	3	A ← input data
OUT	2	3	Output data ← A
			For the above instructions byte 2 at T_1 via the address bus the I/O device is addressed. For IN data is loaded via DBIN strobe. For OUT A is made available when WR signal is 0.
EI	1	1	Interrupt enable
DI	1	1	Interrupt disable
			The interrupt enable flip-flop is set or reset by the above instructions.

Stack Instructions

Mnemonic	*Bytes*	*Cycles*	*Operation*
LXI SP	3	3	SP_1 ← B_2, SP_{11} ← B_3. Load byte 2 and byte 3 of instructions into lower and upper order stack pointer.
PUSH PSW	1	3	[SP − 1] ← A, [SP − 2] ← F, SP = SP − 2 Save A and F (5 flags) into stack addressed by SP. SP is decremented by 2.

PUSH B	1	3	SP −1 ← B, SP − 2 ← C, SP v SP − 2
PUSH D	1	3	SP − 1 ← D, SP − 2 ← E, SP = SP − 2
PUSH H	1	3	SP − 1 H, SP − 2 ← L, SP = SP − 2
POP PSW	1	3	F ← SP, A ← SP + 1, SP = SP + 2 Restore last values of stack to register addressed by SP into A and F.
POP B	1	3	C ← SP, B ← SP + 1, SP = SP + 2
POP D	1	3	E ← SP, D SP + 1, SP = SP + 2
POP H	1	3	L ← SP, H SP + 1, SP = SP + 2
XTHL	1	5	L ↔ SP, H ↔ SP + 1, SP = SP Exchange registers H and L of stack addressed by SP. SP itself not changed.
SPHL	1	1	SP ← H, L. Transfer H and L into SP.
PCHL	1	1	PC ← H, L. Jump indirect.
INX SP	1	1	SP ← SP + 1
DCX SP	1	1	SP ← SP − 1
DAD SP	1	3	H, L ← H, L + SP. Add SP to H and L. If overflow carry FF set.

Other instructions

LDAX D	1	2	A ← D, E. Load A with M addressed by D and E
STAX D	1	2	D, E ← A. Store Store A in M addressed by D and E.
INX D	1	1	D, E ← D, E + 1, Contents of D and E incremented by 1.
DCX D	1	1	D, E ← D, E − 1.

Mnemonic	*Bytes*	*Cycles*	*Operation*
DAD D	1	3	H, L ← H, L + D, E

There are also similar instructions to all the above for registers B and H.

XCHG	1	1	H ← D, E ← L. Exchange registers H and L with D and E.
DAA	1	1	The 8-bit A containing arithmetic results is converted to second BCD values.
SHLD	3	5	Store H and L in M as addressed by byte 2 af 3 of instructions.
LHDD	3	5	Load H nd L by M as addressed by byte 2 and 3 of instructions.
HLT	1	1	Processor goes to STOP state.

Abbreviations used

A accumulator	B_1 byte 1
M memory	H, L, D, E, B, and C stack registers (indicated also by r).
PC program counter	F flag register
SP stack pointer	A logical AND
	Λ logical EXCLUSIVE-OR
	V logical OR

Appendix E

MC6800 INSTRUCTION LIST

Mnemonic	*Operation*	*Mnemonic*	*Operation*
ABA	Add accumulators	CLR	Clear
ADC	Add with carry	CLV	Clear overflow
ADD	Add	CMP	Compare
AND	Logical AND	COM	Complement
ASL	Arithmetic shift left	CPX	Compare index register
ASR	Arithmetic shift right	DAA	Decimal adjust
BCC	Branch if carry clear	DEC	Decrement
BCS	Branch if carry set	DES	Decrement stack pointer
BEO	Branch if equal zero	DEX	Decrement index register
BGE	Branch if > or = zero	EOR	Exclusive OR
BGT	Branch if > than zero	INC	Increment
BHI	Branch if higher	INS	Increment stack pointer
BIT	Bit test	INX	Increment index register
BLE	Branch if < or -	JMP	Jump
BLS	Branch if lower or	JSR	Jump to subroutine
BLT	Branch if < zero	LDA	Load accumulator
BMI	Branch if minus	LDS	Load stack pointer
BNE	Branch if = zero	LDX	Load index register
BPL	Branch if plus	LSR	Logical right shift
BRA	Branch always	NEG	Negate
BSR	Branch to subroutine	NOP	No operation
BVC	Branch if overflow clear	ORA	OR accumulator
BVS	Branch if overflow set	PSH	Push data
CBA	Compare accumulators	PUL	Pull data
CLC	Clear carry	ROL	Rotate left
CLI	Clear interrupt mask	ROR	Rotate right

Mnemonic	*Operation*
RTI	Return from interrupt
RTS	Return from subroutine
SBA	Subtract accumulators

SBC	Subtract with carry
SEC	Set carry
SEI	Set interrupt mask
SEV	Set overflow
STA	Store accumulator
STS	Store stack register
STX	Store index register
SUB	Subtract
SWI	Software interrupt
TAB	Transfer accumulators
TAP	Transfer accumulators to condition code register
TBA	Transfer accumulators
PA	Transfer condition code register to accumulator
TST	Text
TXS	Transfer index register to stack pointer
TSX	Transfer stack pointer to index register
WAI	Wait for interrupt

Index

A

ARRPA network, 10–12
ASC 11 (see Code)
Abacus, 2
Accumulator (see Register A)
Adder, 52, 65
Addition (see Arithmetic)
Aiken, H., 6
Analog, 5–6
Analyzer
  Digital differential (DDA), 5
  Electronic differential (EDA), 5
  Mechanical differential (MDA), 5
Arithmetic
  Addition, 25, 26, 27
  Comparison, 215
  Complement, 28–31
  Division, 67–68
  Double precision, 68
  Fixed-point, 210–212
  Floating point, 69–72, 210–214
  Multiplication, 67–68
  Subtraction, 67
  (see also Code)
Automatic test, 243–245

B

B-register (see Register B)
Babbage, C., 2
BASIC, 148–151
Bidirectional lines (see Lines)
Binary, 17, 25
Binary Coded Decimal (Abbr. BCD, see Code)
Bit, 40–41
Bulk store
  Charge coupled devices (Abbr. CCD), 98
  Continuous magnetic tape recorder (Abbr. CMTR), 92–94
  Core memory, 62–64
  Disks, drums, 91–95

Floppy disks, 95
Magnetic domains, 94–100
Buses, 65, 196
Bush, V., 5
Bytes, 41

C

Card, Hollerith, 4, 171–175
COBOL
Divisions, 172–173
Examples, 173
Features, 174
General, 171
Code
Alphanumeric, 11, 23
American Standard Code for Information Interchange (Abbr. ASC 11 or USASK 11), 17, 24
Baudot, 17
Binary, 16, 17, 25–28
Binary coded decimal (Abbr. BCD), 17–21
Hexadecimal (Abbr. HD), 17, 36–38
Hollerith, 4
Octal, 35–36
Coefficients (see Arithmetic, floating point)
Command (see Instructions)
Comment (see FORTRAN)
Compiler (see BASIC, COBOL, FORTRAN)
General discussion, 43
Computer, 6, 7
EDVAC, 6
ENIAC, 77–83, 106–111
H316, 181–190
Interdata 7/16, 192–227
MARK 1, 6
Microcomputers, Chapter 12
Minicomputers, Chapters 10, 11
SEAC, 7
UNIVAC 1, 7
Control processing, 8, 9
Conversion
Analog/digital, 100
Digital/analog, 101

D

Data, 49
Decimal, 15
Decoder, 51, 72, 73
Demultiplexer, 101–103
Disney World, 244, 245
Displays
Cathode ray tube, 90
Panel, control, 185–187, 196, 197
Plotter, 90
Printer, 88–91
Division (see Arithmetic)

E

Encoder, 101
Exclusive OR (see Gates)
Exponent (see Arithmetic, Floating point)

F

Flip-flops (see Logic)
Floating-point (see Arithmetic, Floating point)
Flow chart, 116–119
FORTRAN
  Arithmetic, 160–161
  Control (GO TO, IF, DO, etc.), 161–165
  Conversational, 169–170
  Examples, 165
  Limitations, 170–171
  Number representation, 156
  Specifications, 165–167
  Subprograms or subroutines, 167–169

G

Gates
  AND, 47
  Exclusive-OR, 47
  Inverter, 47
  NAND, 47
  NOR, 47
  OR, 47
Generator (see Programming)

H

Half word (see Byte)
Hexadecimal (Abbr. HD) (see Codes)
Hollerith, H., 4
Honeywell H316 (see Computers, H316)

I

Index register (Abbr. I-R, see Registers, index)
Input/output
  BIO, 54, 83, 84
  Bidirectional, 58–59
  DIO, 53, 78–82
  Parallel, 53–59
  Serial, 40
  Unidirectional, 78–81
Instructions
  Arithmetic, 210
  Branch (see Jump)
  Floating point, 210–213
  Generic, 107
  Input/output, 109
  Jump, 199, 205–209
  Memory reference, 107–108
  Microinstructions, 77

Modifications: Indexing, 125; Indirect addressing, 125
Multiple address, 112–113
Optional, 109–110
Shift, 107–108
Single address, 111–112
Special devices, 109–110
Integrated circuits (Abbr. I-Cs), 45, 46
Interdata 7/16
General, 192–194
Instructions, 197–220
Operation, 192–197
Peripherals, 194, 226, 227
Program examples, 221–226
Program status word (Abbr. PSW), 194–195, 215
Interface (see Input/output)
Analog, 102–103
Digital, 103
Pulse train (incremental), 103
Interpreter
BASIC, 149–151
General, 148, 178
Interrupts
Microcomputer, 234–240
One line, 65, 82, 84
Priority, 183–184

J

Jump (see Instructions, Jump or Branch)

K

Kelvin, Lord, 5
Keypunch, 2–5

L

Languages
Machine, 101
Mnemonic, 101, 114
Object, 101
Programming: Analysis, 113–114; Examples, 115–117; Steps, 115–117
Software: Business, 147; Decision steps, 119–121; Loading, 145; Manufacturer supplied, 145–147
Source, 101
Large scale integrated circuits (Abbr. LSICs) (see I-Cs)
Level changers, 57
Line drivers, 58
Line receivers, 58
Lines, interface
One way, 58
Two way, 59
Lines, twisted pair, 57
Loading peripherals, 87
Card, 87
Cassette, 97
Loading programs
Absolute, 145
Initial, 145
Relocatable, 145

Logic
Combinational, 51–52
Conversion of logic types, 54, 55
Flip-flops, 48
Counters, 49, 50

M

Macroprogramming, 77, 139
Medium level integrated circuits (Abbr. MSICs) (see I-Cs)
Memory
Core, 62–64
Function, 61, 62
Semiconductor, 64, 65
Microcomputers
Applications, 241
Basics, 229, 231, 232
8080A, 233–235
M6800, 235–237
16 bit, 237, 238
Stack operation, 230, 231
Microprogramming, 77
Minicomputers
H316, Chapter 10
Interdata 7/16, Chapter 11
of microprocessor components, 238–241
Multiplexers, 101
Multiplication (see Arithmetic)
Multiplier operation, 67, 68
Multivibrators, 50

N

NAND (see Gates), 47
Napier, J., 1
NOR (see Gates), 47
Numbers systems (see Codes)

O

Object code (see Languages)
Octal (see Codes)

P

Pascal, B., 1
Peripheral control unit (Abbr. PCU), 78–83
Plotter (see Displays, Plotter)
Pointer, stack (see Microcomputers, Stack)
Printer (see Displays, Printer)
Program counter (Abbr. P-C, same as I-C)
Program status word (Abbr. PSW), 194, 195, 214–218
Programming
Assembler, 129–132, 135–139
Business, 146, 147
Compilers, 129–132, 140–142
Control, 8, 9
Diagnostic (debugging), 133, 134
Generators, 133
Loading programs, 145
Machine code (see Languages)

Meaning of terms, 125, 126
Object code (see Languages)
Simulators, 129
Subprograms, 142, 143
Subroutines, 122
Utility, 128, 129

Q

Quotient, 28

R

Read access memory (Abbr. RAM), 64, 65
Read only memory (Abbr. ROM or pROM), 57
Registers
A, 65, 151
B, 65, 151
Double length, 67, 181, 198
I-C, 75, 76
Index, 63, 74, 75
Input/output (Data buffer of Fig. 12-4), 236, 238
Memory address register (Abbr. MAR), 63, 66
Memory buffer register (Abbr. MBR), 63, 66
Quotient (Abbr. Q), 65
Shift, 56
Stack, 230–231

S

Sectoring, 107
Shift register (see Registers)
Stack pointer (see Pointer)
Stack registers (see Registers)
Stock market system, 249

T

Teletype, 88, 89
Time-share, 7
Timing, 76, 77
Truth table, 47

U

Utility systems, 245–247

V

Variant bits, 123
Vectored interrupts (see Interrupts, priority)
von Leibniz, Baron, 2

W

Wall Street stock monitoring, 249
Words: 16-bit, 107–113, 198; 32-bit, 198